CS 150 Logical Operations

Goals

Now that you have had some experience with assembly language, lets move on to something a little more challenging.

So, let's...

- Learn how to write subroutines.
- Learn how to use the run-time stack.
- Learn how DeMorgan's Law works.
- Learn how to convert a binary character string to a binary number.
- Learn how to convert a binary number to an ASCII binary and decimal character string.

Lab Description...

1. To successfully complete this lab:

   "Write a LC-3 assembly language program that prompts the user for two 2's complement binary numbers, computes a bit-wise "NAND", "**NOR**", and "**XOR**" of those two numbers, and outputs the results in binary and signed decimal to the console."

2. You are required to write several subroutines in completing this lab:
   1. A subroutine that reads binary ASCII characters (**0**'s or **1**'s) from the keyboard and stores them in a memory buffer.
   2. A subroutine that converts a memory buffer of binary ASCII characters into a 16-bit 2's complement binary number.
   3. A subroutine that converts a 16-bit binary number to ASCII binary characters that are output to the console.
   4. A subroutine that converts a 16-bit 2's complement binary number to ASCII signed decimal characters that are output to the console.
   5. A subroutine for each of the **NAND**, **NOR**, and **XOR** functions.

   ***ALL** subroutines must save and restore (using the run-time stack, pointed to by **R6**) any registers modified by that subroutine. (If a register is not changed by the subroutine, it doesn't need to be saved/restored.)

   ***ALSO** the stack must be used to pass subroutine arguments if two or more arguments are required. If only one argument is used in the subroutine, it can be passed simply by putting it into a register.

After being prompted at the console, the user enters two character strings representing 16-bit 2's complement binary numbers at the keyboard. These ASCII characters need to be converted into 16-bit 2's complement binary numbers which are native to the LC-3 computer. Your program then calculates and prints to the console the results of bit-wise **NAND**, **NOR**, and **XOR** operations of the two binary numbers in binary and signed decimal notation.

3. The basic algorithm for this lab might be:
    1. Initialize/setup character buffers and variables (ex: LEA R6,Stack)
    2. Prompt for the first binary number (ex: LEA R0,MES1; PUTS).
    3. *Call a subroutine (JSR) to read into a buffer a string of valid ASCII characters (**0**'s and **1**'s).
    4. *Call a subroutine (JSR) to convert the ASCII characters in the buffer to a 16-bit 2's complement binary number.
    5. Prompt for the second binary number (ex: LEA R0,MES1; PUTS).
    6. *Call a subroutine to read into a buffer a string of valid ASCII characters (**0**'s and **1**'s).
    7. *Call a subroutine to Convert the ASCII characters in the buffer to a 16-bit 2's complement binary number.
    8. *Call a subroutine (JSR) to **NAND** the two numbers together.
    9. Output a "**NAND** result" message to the console.
    10. *Call a subroutine (JSR) to Output the 16-bit binary **NAND** result to the console.
    11. *Call a subroutine (JSR) Output the 16-bit signed decimal **NAND** result to the console.
    12. *Call a subroutine to **NOR** the two numbers together.
    13. Output a "**NOR** result" message to the console.
    14. *Call a subroutine to Output the 16-bit binary **NOR** result to the console.
    15. *Call a subroutine to Output the 16-bit signed decimal **NOR** result to the console.
    16. *Call a subroutine to **XOR** the two numbers together.
    17. Output a "**XOR** result" message to the console.
    18. *Call a subroutine to Output the 16-bit binary **XOR** result to the console.
    19. *Call a subroutine to Output the 16-bit signed decimal **XOR** result to the console.
    20. Go back to step 1.

    *These are to be subroutines!

4. The following guidelines should be followed in designing your program:
    - Your input character buffer routine should only accept binary ASCII characters ('0' or '1') from the keyboard and return when an "**Enter**" character (**x0A**) is entered.
    - Valid binary ASCII characters should be echoed back to the console as they are entered.

    - Any character other than binary characters should echo to the console as a bell character (**x07**).
    - A maximum of 16 ASCII binary characters can be entered at a time. Any more should result in the bell character (**x07**) being output to the console.
    - If less than 16 ASCII binary characters are entered, the conversion subroutine should sign extend the most significant bit to 16-bits.
5. An example output of your program should be:

   ```
   Enter A: 0110010111
   Enter B: 10100
   A NAND B = 1111111001101011 (-405)
   A NOR B = 0000000000001000 (8)
   A XOR B = 1111111001100011 (-413)

   Enter A:
   ```

Before You Begin...

1. Decide on your approach to solving the problem... You might want to consider:
    - What information will my program prompt for?
    - What is considered valid input?
    - What should my output look like?
    - How will I test my program?
2. Layout your program structure. Use a systematic decomposition approach to your problem until you arrive at LC-3 assembly instruction.
3. Use your favorite text editor to create and edit your assembly program.
4. Finally, when you have decided what, where, and how, proceed with your implementation.

Hints and Suggestions

- Make sure you save your assembly files with the "**.asm**" extension or they will not be assembled by the assembler.
- Include comments that explain the abstract flow of your program as you write your program. Coming back later and adding comments loses much of the value comments provide in program development. Essentially, it will save you time in the long run.
- Use meaningful names for labels and variables.
- The assembly directive "**PUTS**" is a trap instruction that will print all characters in a string until reaching a **NULL** value. To use this directive, make sure your strings end with a **NULL** character.
- Do not use the "**IN**" trap instruction for this program, since it prints out a ':' and echoes the input back.
- Remember, "**NAND**", "**NOR**" and "**XOR**" are not part of the LC-3 ISA, so you'll need to create algorithms that will use **AND**'s, **ADD**'s, and **NOT**'s to implement these operations.

Let's Get Started...

Your algorithmic solution has to be systematically decomposed down to the level that is "syntactically correct" for a LC-3 assembler. This is not a hard process, but one that requires exactness.

Carefully follow the steps below for a successful completion of this lab.

Step 1

After you have systematically decomposed your solution to the problem down to the LC-3 ISA abstraction level (assembly), students usually choose one of the following programming methods:

1. Type the whole program into your editor, get it to assemble, load it into the simulator and execute the program. Doesn't work, something's wrong, haven't a clue where to start... try this... try that... 15 hours later and still in the lab...
2. **OR**, they incrementally develop their program by writing a small "working" program (even as small as **HALT** (TRAP x25)) and then after that, they never "break" it. They add a few lines, assemble, and test. Add a few more lines of code, assemble, and test. Soon, they're done, passed off, and out of the lab in an hour.

It's your choice, but please CHOOSE WISELY!

Begin by entering a small program into your editor that you know will work. Add the assembler directives to load your program at memory location **x3000** (**.orig**) and put an end assembly directive (**.end**) at the end of your program. Save your program in a file with a "**.asm**" extension.

For example:

```
;;*******************************************************
;;          Name: CS150Student
;;    Assignment: Logic Lab
;;     File name: logic.asm
;;
;;  Date Created: 02/01/2009
;; Last Modified: 02/22/2009
;;
;; Program Description:
;;   This program prompts the user for two 2's complement
;;   binary numbers, computes a bit-wise NAND, NOR, and
;;   XOR of those two numbers, and outputs the results in
;;   binary and signed decimal to the console.
;;*******************************************************

;;*******************************************************
;; main program
        .ORIG x3000
```

```
START    LD    R6,STACK        ; set the stack pointer
         LEA   R0,msgA         ; prompt for A
         PUTS
         JSR   GetBStr         ; get binary string from keyboard
         LEA   R0,msgB         ; prompt for B
         PUTS
         JSR   GetBStr         ; get binary string from keyboard
;
         LEA   R0,msgNAND      ; output NAND message
         PUTS
         LEA   R0,msgNOR       ; output NOR message
         PUTS
         LEA   R0,msgXOR       ; output XOR message
         PUTS
;
         BR    START           ; loop back to the beginning

;;*********************************************************
;; main constants and strings
STACK   .FILL  STACKB          ; bottom entry of stack
msgA    .STRINGZ "\nEnter A: "
msgB    .STRINGZ "\nEnter B: "
msgNAND .STRINGZ "\nA NAND B = "
msgNOR  .STRINGZ "\nA NOR B = "
msgXOR  .STRINGZ "\nA XOR B = "

;;*********************************************************
;; get ASCII binary string subroutine
GetBStr ADD    R6,R6,#-1       ; push R7 (return address)
        STR    R7,R6,#0

        LEA    R0,MES1         ; point to message
        PUTS                   ; output to console
        LD     R0,VALUE        ; return a '10'

;; restore registers used in subroutine
        LDR    R7,R6,#0        ; pop R7 (return address)
        ADD    R6,R6,#1
        RET                    ; return

VALUE   .FILL  #10             ; return value
MES1    .STRINGZ "GetBStr - To Be Implemented!"

;;*********************************************************
;; Stack area
        .BLKW  50              ; stack size of 50 words
STACKB  .FILL  x0000           ; Base of stack memory
        .END
```

BTW: Your assembly program should be well commented.

Step 2

Your program needs to be assembled before it can be "executed" by the LC-3 simulator. Undoubtedly, the first few passes through the assembler will disclose "syntax" errors. Correct these and continue the edit/assemble cycle until you have a "clean" assembly.

Now load your program object file into your simulator and verify that it works (ie. prints the message and halts). **DO NOT** leave **Step 2** until this works! If it doesn't assemble, load, or whatever, make the necessary changes and try again.

Step 3

From here on you should "incrementally" add functionality to your program by making small, meaningful changes to your program. Assemble often to "catch" syntax errors.

After a "clean" assembly, load your program object file into your simulator and verify that it works the way you intended. If there is a problem, you know where to fix it! Do this over and over until the lab is completed!

Work out an incremental approach to completing the lab. You may choose to "stub out" various parts of your program until they can be implemented. This might mean that you "make up" data (like a "canned" input buffer) and then later replace that code with the "real" code. The point is, "**DON'T BREAK YOUR PROGRAM!**".

Here is a suggested list of incremental steps to complete your lab:

1. Implement and test code to output both the request and response messages to the console.
2. Implement code to convert a 16-bit 2's complement number to ASCII binary characters and output to the console. Test your code by loading known values into R0 and calling your subroutine.
3. Implement code to calculate the "**NAND**" value of two binary numbers. Test your code by loading known values into registers R3 and R4 and then calling your binary output conversion routine.
4. Implement and test code to get characters from the keyboard, check for valid binary characters, echo to the console and store the characters in LC-3 memory as an array.
5. Implement code that converts a LC-3 memory array to a 16-bit 2's complement number. Test by calling your routine twice, saving the return values in registers R3 and R4, and then calling your "**NAND**" and conversion subroutines from above.
6. Implement and test code as above for "**NOR**" and "**XOR**" subroutines.
7. Finally, implement and test code to convert a 16-bit 2's complement number to ASCII signed decimal characters and output to the console.

And Finally...

1. Thoroughly test your program. Check the limits of both your input and output routines. Make sure you sign extend numbers correctly. Verify your logical operations. **Don't ask the TA's to debug your program!**
2. You will need to pass off your Conversion and Logic Lab with a TA. The TA will need to see your commented LC-3 assembly source code as well as run your program on the LC-3 simulator. If you are using a Windows system, be prepared to demonstrate your program on your computer.
3. You **ARE REQUIRED** to electronically submit your lab report through blackboard.

Grading Criteria

This lab is worth 30 points as follows:

4 points	Your program prompts the user for two character strings. After completing an operation, your program outputs a response message to the console followed by the operation results (include output to prove this).
5 points	Your input routine accepts only binary characters, terminates input after 16 characters, and correctly converts an ASCII binary string to a 2's complement 16-bit number (include output to prove this).
5 points	The **NAND**, **NOR**, and **XOR** operations are correctly implemented as verified by correct binary outputs (include output to prove this).
4 points	Your convert to signed decimal routine correctly converts a binary number to a signed decimal string, handling negative numbers and suppressing leading zeroes (include output to prove this).
6 point	Your assembly program correctly implements the subroutines and stack as outlined in the lab description. (This includes saving/restoring registers on the stack and passing arguments correctly using the stack.) Required Subroutines (All Following the protocol outlined): • Input from the I/O console, a 16 binary character string into a memory buffer. • Convert a memory buffer string to a 16-bit 2's complement number. • Convert a 16-bit binary number to an ASCII binary string and output

to the console.

- Convert a 16-bit 2's complement number to an ASCII signed decimal string and output to the console.
- Compute the **NAND** of two numbers.
- Compute the **NOR** of two numbers.
- Compute the **XOR** of two numbers.

6 point	Your assembly program uses a consistent, readable coding style and has relevant comments explaining your program's operation.

In addition to the above points, the following bonus apply:

+2 point	Passed off with a TA at least one day early or passed off to Daniel Evans or Yi Guo
+4 points	In addition to binary and signed decimal outputs, your program also outputs hexadecimal results for each of the operations.