Midterm II Practice Problems

Due: Thursday, May 19th

A few of these questions may appear on or help you prepare for the exam on **Thursday, May 19th**.

1. Consider the following LC-3 assembly program:

```
1          .ORIG x3000
2          LEA R0, LABEL
3          AND R1, R1, #0
4          STR R1, R0, #5
5          TRAP x22
6          TRAP x25
7
8  LABEL   .STRINGZ "Hello, world!"
9          .END
```

   What is the output of this program?

   This program loads into `R0` the address of the 'H' in "Hello, world!". It then sets the memory location 5 addresses after the 'H' to 0, thereby replacing the ',' with a null character. Since `TRAP x22` invokes `PUTS`, which stops once it reaches a null character, the program outputs "`Hello`".

2. Consider the following LC-3 assembly program:

```
1          .ORIG x5000
2  FOO     LD  R0, BAY
3          OUT
4
5  BAR     .BLKW #2
6  BAY     .FILL x61
7          .END
```

   (a) Construct the symbol table for this program.

Symbol	Address
FOO	*0x5000*
BAR	*0x5002*
BAY	*0x5004*

   (b) There is an error in this program — will it be detected by the assembler?

   No. The error is that the program lacks a `HALT`. Nevertheless, it is valid assembly syntax. The assembler can still translate this program into valid machine code, and only at runtime will an error occur.

3. Consider the following LC-3 assembly files, both of which place `0x000F` at memory location `0x4000`:

```
1          .ORIG x3000
2          AND R0, R0, #0
3          ADD R0, R0, #15
4          STI R0, ADDR
5          HALT
6
7  ADDR    .FILL x4000
8          .END
```

```
1          .ORIG x4000
2          .FILL #15
3          .END
```

What is fundamentally different about these two approaches to the same task?

The first uses machine instructions to programmatically calculate and then store the value in memory once it is run. The second hardcodes the value directly into memory as soon as it is loaded.

4. Consider the following LC-3 assembly subroutine:

```
1  FOO     ST  R0, SAVER0
2          AND R0, R0, #0
3          JSR BAR
4          LD  R0, SAVER0
5          RET
6
7  SAVER0  .FILL x0000
```

What is the error in this subroutine?

R7 will be overwritten by the call to BAR, causing the subroutine to infinitely "return" to itself.

5. Recall that `GETC`, the service routine `TRAP x20`, reads a single typed character from the keyboard, but does not echo it to the terminal.

   (a) Develop a subroutine to read a typed character, echo it to the terminal, and return it in `R0`.

```
1  ECHO    GETC
2          OUT
3          RET
```

   (b) What instructions should be used to call this subroutine?

   Either JSR or JSRR to the address of ECHO

   (c) What changes would need to be made to call this subroutine as `TRAP x26`?

   Memory location 0x0026 must be manually set to the address of ECHO, which should be moved to the area of memory reserved for the OS, and the terminating RET must be changed to RTI.

6. How many trap service routines can the LC-3 support?

Each trap vector is 8 bits long, and the Trap Vector Table occupies memory locations 0x0000 to 0x00FF. Both of these factors mean than the LC-3 is limited to 256 traps.

7. What problem could occur if an LC-3 program does not poll the KBSR before reading from the KBDR?

The program could read a character that has already been read.

8. How does the performance of an interpreted program compare to that of a compiled program?

   Interpreted programs are typically slower than compiled programs. Translating high-level expressions into machine instructions at runtime adds overhead, whereas a compiled program is already composed of machine instructions at runtime.

9. Suppose `int a = 6` and `int b = 9`. What do each of the following expressions evaluate to?

   (a) `a | b`

   0110 OR 1001 is 1111: 15

   (b) `!(a + b)`

   $6 + 9$ *is* 15*, which is "true", and "not true" is "false":* 0

   (c) `a = b = 5`

   Assignment evaluates to the value of the right-hand-side and has right-to-left associativity: the values of `a` *and* `b` *are set to* 5*, and the expression itself evaluates to* 5

   (d) `++a + b--`

   The value of `a` *is incremented to* 7 *and evaluates to* 7*; the value of* `b` *is decremented to* 8 *but evaluates to* 9*:* $7 + 9 = 16$

10. Consider the following C fragment:

```
1  long int i, j, count = 0;
2
3  scanf("%d", &i);
4  for (j = 0; j < 32; j++) {
5      if (i & (1 << j)) {
6          count++;
7      }
8  }
9
10 printf("%d\n", count);
```

    What does this fragment do?

    Here, `i` *contains a* 32*-bit integer scanned from user input. The "*`for`*" loop iterates* 32 *times. On each iteration,* `1 << j` *evaluates to a* 32*-bit mask containing a '*`1`*' in the* j^{th} *bit and a '*`0`*' in all other bits. Applying that mask results in a truthy value if and only if* `i` *contains a '*`1`*' in its* j^{th} *bit.*

    Thus, this fragment reads an integer from user input and prints the number of bits set to '`1`*' in its signed, two's complement binary representation.*

11. Consider the following C program:

```c
#include <stdio.h>

void foo(int);

int main() {
    int z = 2;

    foo(z);
    foo(z);

    return 0;
}

void foo(int z) {
    printf("%d\n", z);
    z++;
}
```

What is the output of this program?

2
2

12. Recall that the runtime stack contains stack frames, and, by convention, an LC-3 stack frame contains a function's arguments, return value, return address, dynamic link, and local variables.

(a) Are the arguments pushed by the caller or the callee?

The caller — the callee cannot possibly know the intended values of its arguments.

(b) What is the purpose of the return address?

The return address saves the caller's PC.

(c) What is the purpose of the dynamic link?

The dynamic link saves the caller's frame pointer.

(d) Why are local variables stored on the stack?

So that their memory will be deallocated when the function returns and its stack frame is popped, and so that each function application has its own independent set of local variables.

13. Suppose that a C function `foo` has been compiled into LC-3 assembly, producing the following initial assembly instructions:

```
FOOFN   ADD R6, R6, #-1
        ADD R6, R6, #-1
        STR R7, R6, #0
        ADD R6, R6, #-1
        STR R5, R6, #0
        ADD R5, R6, #-1
        ADD R6, R6, #-4
```

How many local variables are declared at the beginning of `foo`?

Space for 4 local variables has been pushed onto the stack.