This lab covers the material in lectures four through eight.

If you're unsure about a result, be sure to explain your thinking. In general, you should show your work. A well-reasoned but wrong answer (for example with a minor error) will receive more points than a wrong answer with no context at all.

Problems (50 points total)

1. Consider a pointer in C, e.g. a variable declared like `char* x`. How should the system encode this pointer? That is, what underlying representation should it use? How big should the representation be (i.e. how many bits)?

   It should be an unsigned integer since it represents a memory address (we can’t have negative memory addresses). How big it is depends on how many addresses we need to represent! We have to fix our address size to capture a reasonable amount of memory. 64 bits will allow us to address $2^{64}$ bytes of memory, which is much more than we can put in a computer currently.

2. Convert the following decimal numbers to 8-bit, unsigned binary integers. Show your work.

   a) 33
      
      \[ 00100001_{2}. \]
   b) 87
      
      \[ 01010111_{2}. \]
   c) 122
      
      \[ 01111010_{2}. \]
   d) 223
      
      \[ 11011111_{2}. \]
3. Assume a raw digital image with a resolution of 3480x2160 (4K UHD). Also assume 8-bit color depth and that we’re using a 3-channel (RGB) scheme to represent pixels.

a) Ignoring image metadata, how big will an image be in bytes?

\[ 3480 \times 2160 \times 3 = 22550400 \text{ bytes (21MB)}. \]

b) How many unique colors can we represent with one pixel?

\[ 2^{24} \text{ (16 million)}. \]

4. Assume CMOS transistors.

a) Write out the truth table for a 3-input NAND.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>\text{NAND}(A,B,C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

b) Draw a CMOS transistor circuit that implements it. Note the total number of transistors you use.
c) Now write out a truth table for a 3-input AND.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>$\text{AND}(A, B, C)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

d) Draw the CMOS transistor circuit for this truth table. Don’t forget to adhere to the pMOS/nMOS connection rules! **Hint: you’ll need to add inverters to inputs.**
e) Instead of the above, draw an AND circuit that connects an inverter to the output of your previous NAND.
f) How many transistors have you saved by using the inverted NAND gate?

_You should have gone from a 18T (or 12T if you reused your inverted inputs) to an 8T circuit, thereby saving 10 (4) transistors._

5. We saw that an XOR gate was a real pain to translate directly to CMOS. We’re going to apply some trickery to fix that problem, by recalling that a 2-input NAND gate only requires 4 transistors. Recall that I told you that NAND is a _universal_ gate, meaning we can implement _any_ logic function using only NAND gates.

a) First draw out the truth table for XOR. Assume we’ll call the two inputs X and Y.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X ⊕ Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

b) Translate this truth table to a logic function in terms of X and Y written in DNF form (disjunction of conjunctions).
\( XY + X\bar{Y} \)

c) Translate this function directly into a gate-level circuit (using OR gates and AND gates).

d) To convert this circuit into NANDs, first start with your AND gates near the inputs. Convert them into NANDs by adding a bubble on their output. To make this circuit produce the same outputs, you’ll have to add more bubbles. Show the new circuit with all added bubbles.

e) There’s one step left. We have to get rid of that last top-level OR gate. Show how we turn this into a NAND. (Hint: draw out the truth table for an OR gate and invert its inputs, leaving the output column alone. What truth table does this give you?). Show your final gate diagram.

f) Translate this gate diagram into a logic formula.

\( ((XY))((\bar{X}\bar{Y})) \)

g) Prove that this is in fact the same logic formula you started with in part (b) (Hint: you’ll want to use De Morgan’s law judiciously).
\[(XY)(XY) = (X + Y)(X + Y) = (X + Y) + (X + Y) = XY + XY\]

6. I bet you thought you were done with XOR. Nope, sorry. XOR is magical, so I’m going to force feed it to you. Feast on the magic and thank me later.

a) Let us begin with two 8-bit binary strings, A and B. Assume at first \(A = 01001010\) and B has the same value. Take A and B and XOR them together in a bitwise fashion (XOR each bit of A with each bit of B. What bit string do you end up with?

\[00000000\]

b) Now keep A the same and change B to \(01001110\) (we’ve flipped one bit of B). XOR them together again. What do you get now?

\[00000100\]

c) Now set B to \(\neg A\). XOR them together. What do you get?

\[11111111\]

d) What conclusions can you draw from this?

If two binary strings differ, we’ll get a non-zero result when we bitwise XOR them. There will be 1s in the positions where they differ.

e) Using the conclusions you’ve drawn, outline a procedure for how you’d calculate how far two binary strings are from one another, namely how many bit positions differ between the two strings? Pseudocode, C code, or a summary are fine here.

We XOR the two strings together, and then count the number of 1s in the result, giving us the edit distance. There are several ways to do this, but the most straightforward:

```c
unsigned char x = a ^ b; // a XOR b (a and b are inputs)
unsigned differ = 0;
for i = 0; i < 7; i++) {
    differ += (x >> i) & 1;
}
```

f) Let’s look at just A now. What we’re going to do is take the cumulative XOR of the bits of A, from left to right. We can write this as \(A_7 \oplus A_6 \oplus \ldots \oplus A_0\). What is the result?

\[1\]

g) Change A to 11001010. What is the result now?

\[0\]

\[\text{This is often called the edit distance of two strings, and is important in information theory.}\]
h) Try changing A around in other ways. See how it changes the result. Now, how can we interpret the result? I.e., what does the result mean with respect to the binary string A?

*Our XOR bit is counting the number of 1s in our binary string by flipping back and forth. You should have observed that if we have an even number of 1s we end up with 0 as our result; if we have an odd number of 1s, we get 1. Another way to describe this is the sum of the bits in the string modulo 2.*

i) I want to send my 8-bit string over a network to another computer. Networks are imperfect, however, and they may cause the data to be corrupted. Tell me (using the intuition you developed above) how I could detect such a corruption.

*Send the cumulative XOR along with the data. When it gets to the other computer, we perform the same procedure (computing another cumulative XOR). If we get the same bit, we can assume either (1) no errors occurred or (2) some even number of errors greater than 1 occurred. If we assume that it is fairly unlikely for two independent errors to occur in one transmission, we can use this as a reasonable mechanism to detect bit errors.*

---

**Coding Assignment (50 points total)**

Get the lab2 code by logging into fourier and running the following:

```bash
$> cp /home/khale/HANDOUT/lab2.tgz .
$> tar xzvf lab2.tgz
$> cd lab2
$> make
```

You’ll see a few extra files this time. You can see how to run the lab2 program by running it without any arguments:

```bash
$> ./lab2
```

Your job here is to fill in the functions in lab2.c which have *FILL ME IN* comments written in them. See the comments in the code for further instructions. I’ve also included a test harness for you to test your code. You can run it like so:

```bash
$> make test
```
# Testing lab2...
Test #0 - [FAIL]
Test #1 - [FAIL]
Test #2 - [FAIL]
Test #3 - [FAIL]
Test #4 - [FAIL]
Test #5 - [FAIL]
Test #6 - [FAIL]
Test #7 - [FAIL]
Test #8 - [FAIL]
Test #9 - [FAIL]
Test #10 - [FAIL]
Test #11 - [FAIL]
Test #12 - [FAIL]
0 out of 13 test cases passed

Notice that the code will initially fail all the test cases. Your job is to make it pass them all. These are example cases, and I’ve included them in a separate file called examples. You are free to modify this file and add your own tests. The format is:

[option] [input] [expected output]

For example, to test my program with the -f flag (which invokes the bit flipping routine), I can include a line like this:

f 0x0000 0xffffffff

And it will add another test. You can then rerun make test to see whether or not your program passes it. Note that we will be using different test cases than the ones in this example file, so make sure to add your own tests to convince yourself that your code works!

DO NOT MODIFY THE test_harness SCRIPT! Or, do so at your own risk.

Hand-in Instructions

Make sure to put your name on your submission. Submissions without names will be given zero points! For code, this means put a comment at the top of your C file with your name on it.

For the code, you must hand it in digitally. I’ve made this a bit easier this time. Once you’re happy with your code, in the directory where your code is, run the following:

```
$> make handin
```

For the problems, the following still apply:

**Physical** : If you’re submitting a written copy, hand it to one of the TAs or to the instructor. You can also leave it in the instructor’s mailbox in the CS department office, but make sure to get it time stamped when you do (see the “Submitting Work” section of the syllabus).
Digital: If you would like to submit an electronic copy, note that I will only accept PDF files (no Word docs please). Again, see the “Submitting Work” section of the syllabus. Please do not take a poorly lit picture of your assignment. Your grade will suffer commensurately with our inability to read your work. Once you have a PDF, you should submit it on fourier. You should name your file yourid-lab2.pdf where yourid is the thing in front of the @hawk.iit.edu in your e-mail address.

You can first get your PDF (for example, for me it might be called kh123-lab2.pdf) onto fourier like so:

```
[me@mylocalmachine]$ scp kh123-lab2.pdf kh123@fourier.cs.iit.edu:
```

Then you can login to fourier via ssh and submit it:

```
[kh123@fourier]$ cp kh123-lab2.pdf /home/khale/HANDIN/lab2
```