"Beware of bugs in the above code; I have only proved it correct, not tried it." – Donald E. Knuth

Learning Objectives

1. Combinational logic design.
2. Using bitwise logical and shifting operations in a high-level language like C++.

Work that needs to be handed in

1. modify `keypad.v` as described in the **Keypad** section. We've provided an exhaustive test suite in `keypad_tb.v`.
2. modify `extractMessage.cpp` as described in the **Steganography** section.
3. modify `countOnes.cpp` as described in the **Count Ones** section.

Compiling, Running and Testing

If you're on EWS, you'll need to get your PATH set up to be able to easily access various tools throughout the semester. Run the following command to do so; once again, *this is only necessary on EWS*. Google "shell PATH" and take a look at the source of the script if you're interested in what's happening under the hood.

```
source /class/cs233/setup
```

We have included a `Makefile` in your `Lab2` directory that facilitates compilation. The following examples illustrate its use:

```
make keypad         - compiles the keypad circuit and runs its test bench
make example        - compiles the example BMP program
make extractMessage - compiles the extractMessage program
make countOnes      - compiles the countOnes program
make clean          - removes all compiled files
```

To test message extraction, run the following command, which calls your `extractMessage` function on the test BMP image. A correct implementation should print out
`What do you get when you add two sombreros together?  A sumbrero`

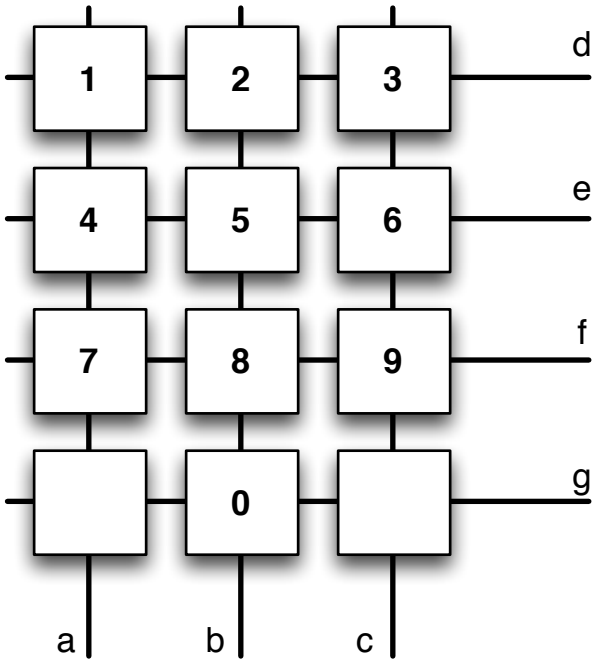
```
./extractMessage picture.bmp
```

To test Problem 2, run the command `./countOnes <number>` with any number of your choice; a correct implementation should print out the number of bits that are 1 in that number. You can use `0b` to prefix binary constants and `0x` to prefix hexadecimal constants. A few examples:

```
./countOnes 55             - should print out 5
./countOnes 0xbaadf00d     - should print out 17
./countOnes 0b10110100     - should print out 4
```

Keypad

Consider the telephone style key pad shown below. This key pad supports numerical buttons `0-9` and has 7 output signals `a-g`. When no buttons are being pressed all of these signals output a 0. When a button is pressed a 1 is output on both the row and the column of the button pressed (*e.g.*, if 6 was pressed c and e would be 1 and all of the other signals would be zero).



Your job is to design a circuit that indicates numerically which button is being pressed (assuming a single button is pressed at a time). Your circuit will take the 7 signals (a-g) as inputs and produce two outputs: the 1-bit signal `valid` indicating that a button was pressed, and the 4-bit signal `number` specifying which number button was pressed (encoded as a binary number).

Write boolean expressions for these signals:

valid =

number[3] =

number[2] =

number[1] =

number[0] =

Steganography

Steganography[1] is the science of concealing a secret message within a "carrier" message in a way that makes it hard to discern the presence of the hidden information (for example by writing in "invisible ink"). Here, we will look at a simple method of hiding a text message within an image, so that the image looks almost unchanged. Given an image (in the BMP format[2]) containing a secret message, your task is to write a C++ program to decipher the message.

Each pixel in a BMP image is encoded with 24 bits as a triple (blue, green, red), where each component is an 8-bit quantity that can take values from 0 to 255. Thus, a red pixel is represented as (0, 0, 255) whereas a mixture of blue and green could be represented as (100, 200, 0). We've given you functions to manipulate BMP files (see `bmp.h` and `bmp.cpp`), as well as an example program (`example.cpp`) illustrating how to use these functions.

Messages are encoded in BMP files according to the following scheme:

(a) First, each letter in the message is encoded as a `char` according to its ASCII code[3]. The message is terminated by a `char` with value 0 (NUL). Thus, the message can be viewed as a sequence of bits.

(b) Next, the message bits are encoded across several adjacent pixels, beginning with pixel (0, 0), followed by pixel (1, 0), (2, 0), ...etc. If the image is k pixels wide, then pixel $(k-1, 0)$ is followed by pixel (0, 1), (1, 1), (2, 1),...

(c) Message bits are stored in the *least* significant bit (LSB) of the `green` component of pixels. For example, a message beginning with the letter 'H' (ASCII code 0x48, i.e. 01001000 in binary) is encoded as follows:

- pixel (0, 0): LSB(`green`) = 0
- pixel (1, 0): LSB(`green`) = 1
- pixel (2, 0): LSB(`green`) = 0
- pixel (3, 0): LSB(`green`) = 0
- pixel (4, 0): LSB(`green`) = 1, . . .

The sample BMP file `picture.bmp` contains the following message encoded according to the above scheme:

```
What do you get when you add two sombreros together? A sumbrero
```

This file can be found in the course SVN under `_shared/Lab2`, and should be automatically downloaded the first time you run the Makefile.

What you need to do

The C++ function `extractMessage` in the file `extractMessage.cpp` accepts one parameter (a BMP object); complete it so that it extracts the message hidden in this BMP (**NOT including the terminating NUL char**) according to the scheme just described and returns it in the provided string variable. To do this, your code must use bitwise operations in C++. Do note that your function should stop extracting the message once it reaches the terminating NUL char.

[1] Steganography: http://en.wikipedia.org/wiki/Steganography
[2] BMP file format: http://paulbourke.net/dataformats/bmp/
[3] ASCII codes: http://www.asciitable.com

Count Ones

In this problem, you will write an efficient C++ function to count the number of bits that are equal to one in an unsigned integer. Here is a simple (but *inefficient*) way to solve the problem:

```
unsigned countOnesDumb(unsigned input) {
  int i, count;
  count = 0;
  for (i = 0; i < 8 * sizeof(unsigned); i++) {
    if ((input & 1) != 0) {
        count++;
    }
    input = input >> 1;
  }
  return count;
}
```

This is inefficient, because it takes $O(n)$ operations[4] on n-bit numbers to count the number of ones in an n-bit integer. We want you to write a function that performs only $O(\log_2 n)$ operations on n-bit numbers for this task.

Here is a description of an efficient algorithm for this problem (you can assume that integers are 32 bits long):

(a) Treat each integer as 32 1-bit counters. Pair adjacent counters off, and add each pair. The result will be 16 2-bit counters.

(b) Take these 16 2-bit counters and pair adjacent counters off. Add the pairs of counters to produce 8 4-bit counters.

(c) Keep pairing and adding in this fashion until you get a single 32-bit counter, which will contain the result.

Is this really efficient? Think about step 1 of this algorithm. For each pair of bits, we need to perform one addition operation - this means $n/2$ additions for an n-bit number. For step 2, we need to perform one addition operation for each of the 16 2-bit counters - this means $n/4$ additions are required for this step. We can continue this trend until we are left with a single 32-bit counter. After all steps are complete, we are left with an algorithm that requires $O(n)$ operations!

The trick to achieve $O(\log_2 n)$ is to do all these additions together (in parallel). In what follows, we'll show you how to do this with an example. To keep things simple, we'll work with 8-bit integers.

What you need to do

The C++ function `countOnes` in the file `countOnes.cpp` accepts an unsigned integer as the input; complete it so that it efficiently counts and returns the number of occurences of 1 in the binary representation of the input *32-bit* integer **using the algorithm described below, and without using loops or conditionals**.

[4]Recall from CS 125/173/225: $O(n)$ means (roughly) at most $c \cdot n$, for some constant $c > 0$

Doing things efficiently

Suppose the input integer is 10110110. We treat this as eight 1-bit counters. First, let's isolate every alternate counter in this integer. We can do this by AND-ing the number with the bit-pattern 01010101 (i.e. 0x55)

```
     1 0 1 1 0 1 1 0
AND  0 1 0 1 0 1 0 1
--------------------
     0 0 0 1 0 1 0 0    odd counters
```

Next we get the other counters, this time by AND-ing with the bit-pattern 10101010 (i.e. 0xAA)

```
     1 0 1 1 0 1 1 0
AND  1 0 1 0 1 0 1 0
--------------------
     1 0 1 0 0 0 1 0    even counters
```

Now we need to pair-up adjacent counters and add them. We can do this by first RIGHT-SHIFTING the even counters by 1 to align them with the odd counters, and then simply adding the two numbers:

```
     0 1 0 1 0 0 0 1    even counters RIGHT-SHIFTED by 1
  +  0 0 0 1 0 1 0 0    odd counters
--------------------
     0 1 1 0 0 1 0 1
```

This completes step 1 of the algorithm. It takes 4 operations (two ANDs, one RIGHT-SHIFT and one + operation) in the case of 8-bit integers, and will similarly take 4 operations in the case of 32-bit integers.

Moving on to step 2, we need to treat the number 01100101 as four two-bit counters as follows:

```
   0 1 1 0 0 1 0 1
   \ / \ / \ / \ /
 =  1   2   1   1
```

We will need appropriate bit-patterns to obtain alternate counters like before. The odd and even counters after this step (*i.e.*, *after AND-ing with the appropriate bit-patterns*) should be:

```
     0 0 1 0 0 0 0 1    odd counters
```

and

```
     0 1 0 0 0 1 0 0    even counters
```

Again, we align the counters (this time by right-shifting the even counters by 2) and add them:

```
     0 0 0 1 0 0 0 1    even counters RIGHT-SHIFTED by 2
  +  0 0 1 0 0 0 0 1    odd counters
--------------------
     0 0 1 1 0 0 1 0
  =  \     / \     /
        3       2
```

Finally, we treat 00110010 as two 4-bit counters (as shown above) and add them to get the final answer: 5.