Learning Objectives

1. Introduction to Verilog syntax and tools (iverilog, gtkwave)
2. Practice with Verilog coding, testing, and debugging
3. Practice designing combinational circuits using a truth table
4. Combinational logic design.
5. Using bitwise logical and shifting operations in a high-level language like C++.

Setup your Development Environment and Github

Check out our Development Guide: https://cs233.github.io/devenvironmentguide.html

Please follow the instructions at https://cs233.github.io/githubprimer.html to setup your github repository for the course. You will find information on getting the lab files there as well.

Work that needs to be handed in

1. **First deadline:**
   (a) Farmer problem on PrairieLearn.
   (b) `sc2_block.v` on PrairieLearn
   (c) Complete Black Box assignment on PrairieLearn.
2. **Second deadline:**
   (a) `keypad.v`: Modify as described in the **Keypad** section. We've provided an exhaustive test suite in `keypad_tb.v`.
   (b) `extractMessage.cpp`: Modify as described in the **Encryption** section.
   (c) `countOnes.cpp`: Modify as described in the **Count Ones** section.

Part 1

Part 1 will be completed in PrairieLearn. You will be able to download the files you need directly from PrairieLearn or you can get the files and test benches you need from GitHub. Only what is submitted in PrairieLearn will be graded. Hopefully this will give you time to work any kinks in your remote setup before Lab 1 Part 2.

Setting up your team in PrairieLearn

1. Do **NOT** start working on Lab 1 Part 1 until you have decided who you will work with (or whether you will work alone)
2. Log into PrairieLearn
3. Choose one person in your team to make the team. That person should create a group name and then share the join code with the rest of the team (if applicable). Those teammates should enter the join code to join the team. Do **NOT** press "Start assessment" until everyone has joined the group.
4. Once everyone has joined and you press "Start assessment," follow the instructions in PrairieLearn to complete Part 1 of the lab.

Accessing software on EWS

If you're on EWS (or accessing it through FastX), 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
```

Part 2: Compiling, Running, and Testing

Part 2 comprises three problems: `keypad`, `extractMessage`, and `countOnes`. We have included a `Makefile` in your `Lab1` folder that facilitates compilation of these problems. The following examples illustrate its use:

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

To test message extraction, run the command `./extractMessage` after compilation. This command calls your `extractMessage` function on the default test message. A correct implementation should print out

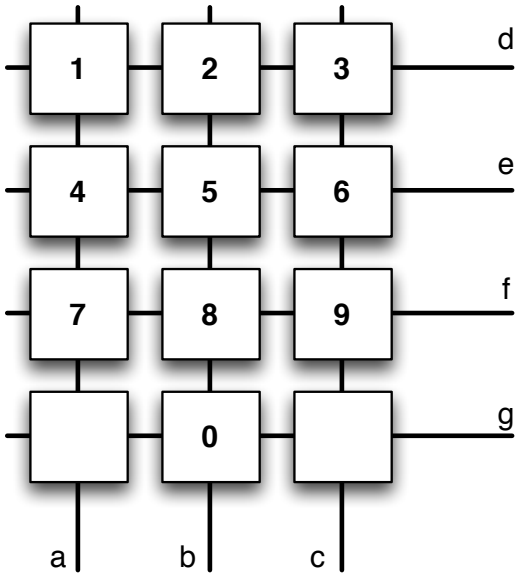
```
Knock knock.
Race Condition.
Who's there?
```

To test Problem 2, run the command `./countOnes <number>` with any number of your choice after compilation. 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 keypad shown below. This keypad 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).



What you need to do: Write Verilog code for a circuit that that indicates numerically which button is being pressed (you may assume 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 numbered button was pressed, and the 4-bit signal `number` specifying which number button was pressed (encoded as a 4-bit unsigned binary number). For example, if button 4 is pressed the output should be valid = 1 and number = 0100.

It may be helpful to first write boolean expressions for the signals:

valid =

number[3] =

number[2] =

number[1] =

number[0] =

Encryption

Encryption[1] is the process of encoding messages or information in such a way that only authorized parties can read it. Here, we will look at a simple method of "encrypting" a text message as an encoded string. Your task is to write a C++ function to decode these encoded messages.

Messages are encoded/decoded according to the following scheme:

(a) First, each character in the message is encoded in 1 byte (8 bits) according to its ASCII code[2]. For encoding/decoding purposes, the length of the message must be a multiple of 8 bytes. The length of the message must be provided to the encoder/decoder along with the message itself.

(b) Each group of 8 bytes is encoded/decoded independently.

(c) Within each group of 8 bytes, we're going to "reflect" the bits, so that the bits of the ith byte of the input becomes the ith bit of each of the 8 bytes of the output, with the LSB of the ith byte becoming the ith bit of the 0th byte of the group. (Alternatively, you could view this as least significant bits (LSbs) of each group of 8 input bytes containing the bits of the first character of the group of 8 output bytes, the second LSbs containing the bits of the second character, ..., the most significant bits (MSbs) of the 8 characters containing the bits of the 8th character.)

As an example, say we're given the message:
`Hello World!` If we represent this as a set of bits in ASCII, pad it with the null character, and perform the scheme above, the transformation will look like this:

H	0	1	0	0	1	0	0	0
e	0	1	1	0	0	1	0	1
l	0	1	1	0	1	1	0	0
l	0	1	1	0	1	1	0	0
o	0	1	1	0	1	1	1	1
	0	0	1	0	0	0	0	0
W	0	1	0	1	0	1	1	1
o	0	1	1	0	1	1	1	1
r	0	1	1	1	0	0	1	0
l	0	1	1	0	1	1	0	0
d	0	1	1	0	0	1	0	0
!	0	0	1	0	0	0	0	1
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0

$\rightarrow$

	o	W		o	l	l	e	H
210	1	1	0	1	0	0	1	0
208	1	1	0	1	0	0	0	0
222	1	1	0	1	1	1	1	0
157	1	0	0	1	1	1	0	1
64	0	1	0	0	0	0	0	0
190	1	0	1	1	1	1	1	0
223	1	1	0	1	1	1	1	1
0	0	0	0	0	0	0	0	0
					!	d	l	r
8	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	1
6	0	0	0	0	0	1	1	0
2	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	1
15	0	0	0	0	1	1	1	1
7	0	0	0	0	0	1	1	1
0	0	0	0	0	0	0	0	0

The transformation for a group of 8 bytes ($8X8$ matrix) is a transpose function, where a bit at

[1] Encryption: https://en.wikipedia.org/wiki/Encryption
[2] ASCII codes: http://www.asciitable.com

location (i, j) is stored at location $(7 - j, 7 - i)$.

The `extractMessage_main.cpp` contains the following message encoded according to the above scheme:

```
Knock knock.
Race Condition.
Who's there?
```

What you need to do

The C++ function `extractMessage` in the file `extractMessage.cpp` accepts two parameters, a pointer to a character array and the length of that character array; Complete it so that it extracts the encoded message according to the scheme just described into the provided character array and returns a pointer to it. **To do this, your code must use bitwise logical and shifting operations in C++. Note that all libraries required for the completion of code have been included in the base code provided to you. Additional libraries included by you will be removed before compilation of code for grading**.

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[3] 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.

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

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. You may not use any additional libraries in your solution. Note that all libraries required for the completion of code have been included in the base code provided to you. Additional libraries included by you would be removed before compilation of code for grading**.

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    (Our starting number)
AND  0 1 0 1 0 1 0 1    (The appropriate mask for the "right" 1-bit counters)
--------------------
     0 0 0 1 0 1 0 0    "right" 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    (Our starting number)
AND  1 0 1 0 1 0 1 0    (The appropriate mask for the "left" 1-bit counters)
--------------------
     1 0 1 0 0 0 1 0    "left" counters
```

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

```
     0 1 0 1 0 0 0 1    "left" counters RIGHT-SHIFTED by 1
  +  0 0 0 1 0 1 0 0    "right" 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. However, because our counters are now two-bits each, the masks and shifts also need to represent that change.

Firstly, isolate the "right" counters:

```
     0 1 1 0 0 1 0 1    (The output of adding both 1-bit counters)
AND  0 0 1 1 0 0 1 1    (The appropriate mask for the two-bit "right" counters)
--------------------
     0 0 1 0 0 0 0 1    "right" counters
```

Likewise, we repeat this step for the "left" counters:

```
     0 1 1 0 0 1 0 1    (The output of adding both 1-bit counters)
AND  1 1 0 0 1 1 0 0    (The appropriate mask for the two-bit "left" counters)
--------------------
     0 1 0 0 0 1 0 0    "left" counters
```

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

```
     0 0 0 1 0 0 0 1    "left" counters RIGHT-SHIFTED by 2
  +  0 0 1 0 0 0 0 1    "right" counters
--------------------
     0 0 1 1 0 0 1 0
  =  \     / \     /
        3       2
```

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