ECE 2372 Modern Digital Design July 7

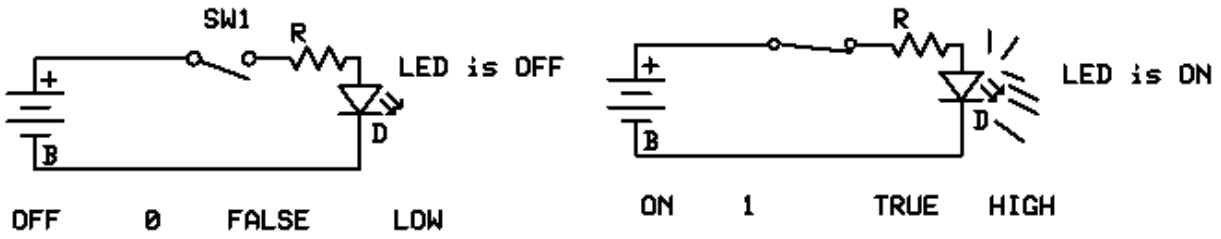
Reading assignment for today: Chapter 1, especially beginning with section 1-3 Number Systems thru to the end of the chapter.

Representing digital information:

Information in modern digital computers is represented using the binary number system. In our everyday life, we routinely use the decimal number system, consisting of ten individual digit values between 0 and 9 for a total of 10 different values for each digit position. In the binary number system, we only have two possible values for a digit position, 0 or 1. This is convenient for electrical or electronic circuits, the two possible values 0 and 1 can be represented by the OFF and ON state of an electric circuit.

In binary, a ZERO represents the OFF state or FALSE (sometimes called LOW state) and is shown as: 0

In binary, a ONE represents the ON state or TRUE (sometimes called HIGH state) and is shown as: 1



In the schematic diagrams above, a battery is use to provide power to an LED (Light Emitting Diode) thru a switch. In the lefthand circuit, the switch is in the OFF state and the LED is OFF representing the ZERO, OFF, FALSE or LOW state. Any of these terms can be used. The righthand circuit has the switch in the ON state, representing ONE, ON, TRUE or HIGH state.

The diagrams above represent the state of a single "bit" of information. A bit is the smallest amount of information we can store in a digital computer, and it can only have one of two values, either 0 or 1 (or FALSE/TRUE, LOW/HIGH, OFF/ON – depending on how we choose to describe it in a particular context).

Binary numbers.

In decimal numbers, each digit position represents a power of 10, increasing as we move to the left. For example, the rightmost digit is the 10^0 position and represents the number of 1's (ones). Likewise the digit one position to the left is the 10^1 position and represents the number of 10's (tens), and the digit position one to the left of the tens position is the 10^2 position of the 100's (hundreds) position. We do something similar in binary. The rightmost position is the 2^0 position or the number of 1's (ones). The position one digit to the left is the 2^1 or the 2's (twos) position, and one digit position further to the left

is the 2^2 the 4's (fours) position. Each digit position further to the left has a weight of 2 times greater as in this sequence 128 64 32 16 8 4 2 1 in the case of an 8-bit (single BYTE) value.

Representing digital information as a single bit is rather limiting. With a single bit, we can only represent the state on one piece of information. It is either TRUE or FALSE, ONE or ZERO, HIGH or LOW, 1 or 0 depending on how we choose to state it. The power in representing information as only 0's or 1's is when those bits are combined into nybbles (4 bits at a time), bytes (8 bits at a time) words (16 or more bits at a time depending on the system).

For example, with only 2 bits (A and B), we can represent a total of 4 combinations: (0 – 3)

2^1 2^0

2's 1's

A	B	decimal equivalent
0	0	0
0	1	1
1	0	2
1	1	3

With only 3 bits (A,B, and C), we can represent a total of 8 combinations: (0 – 7)

2^2 2^1 2^0

4's 2's 1's

A	B	C	decimal equivalent
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

We see that 3 bits provides the ON/OFF state of 3 different items, to describe a total of 8 possible combinations.

Similarly, for any 2^n if n is the number of bits, 2^n equals the total number of possible combinations of those n binary bits. For example, 4 bits yields 2^4 combinations, = 16 combinations. These are given below:

With only 4 bits (A,B, C,and D), we can represent a total of 16 combinations: (0 - 15)

2^3 2^2 2^1 2^0

8's 4's 2's 1's

A B C D	decimal equivalent	hexadecimal equivalent
0 0 0 0	0	0
0 0 0 1	1	1
0 0 1 0	2	2
0 0 1 1	3	3
0 1 0 0	4	4
0 1 0 1	5	5
0 1 1 0	6	6
0 1 1 1	7	7
1 0 0 0	8	8
1 0 0 1	9	9
1 0 1 0	10	A
1 0 1 1	11	B
1 1 0 0	12	C
1 1 0 1	13	D
1 1 1 0	14	E
1 1 1 1	15	F

Note that in hexadecimal, we have 16 combinations, (0 – 15) so to represent the cases beyond 9, we use alphabetic characters A thru F. You may wonder about hexadecimal and how often it is used. Hexadecimal is used extensively in the world of microcontrollers due to convenience. Suppose you needed to remember the following 16 bit binary number:

1110110000110111 This might be a challenge. However, if we break the 16-bit number into 4-bit nybbles: 1110 1100 0011 0111 and then in turn convert each nibble into hexadecimal, we have something easier to handle: 1110 = E 1100 = C 0011 = 3, 0111 = 7, so $1110110000110111 = EC37_{16}$

Most of us can probably remember $EC37_{16}$ easier than we can remember 1110110000110111.

So, for an 8-bit number, how many possible combinations can we have? $2^8 = 256$ combinations, (0-255), 00000000 to 11111111. In hexadecimal this would be from 0x00 to 0xFF (note that 0xFF is an alternative way to indicate hexadecimal in place of FF_{16}. This is treating the binary number as a positive unsigned number. Normally we will treat binary numbers as unsigned positive numbers, but sometimes it is convenient to use a form of binary numbers called 2's complement – this allows us to represent both positive and negative numbers, and we will see later that this is quite convenient.

So, for a 16-bit number, how many possible combinations can we have? $2^{16} = 65536$ combinations, (0-65535), 0000000000000000 to 1111111111111111. In hexadecimal this would be from 0x0000 to 0xFFFF. This is treating the binary number as a positive unsigned number.

Let us consider an 8-bit binary value for a moment. It can have any value from 00000000 to 11111111 or decimal 0 to 255 for a total of 256 possible combinations. The rightmost bit is the Least Significant Bit and can take on the value of 0 or 1 representing zero 1's or one 1 for that digit position. The leftmost digit in an 8-bit binary number is the Most Significant Bit and takes on the value of 2^7 or 128 decimal.

The following will just use 4-bit nybbles for convenience.

```
   0001
  +0001
=  0010
```

Notice that a given digit position in binary can only have the value 0 or 1. If we add 1 to 1, we get 0 with a carry as shown above. 0001 + 0001 = 0010. 1 + 1 = 2, but in binary that is represented 10_2 or if it is obvious that we are working in binary, 1+1 = 10, or if nybbles 0001 + 0001 = 0010.

0 + 0 = 0

0 + 1 = 1

1 + 0 = 1

1 + 1 = 0 with a carry, or 10

For some applications, for example clocks, timers on microwave ovens, and similar systems with digital displays need to display just the numbers 0 thru 9. For convenience, we can represent such numbers as BCD (Binary Coded Decimal). This way we can use a 4 bit nibble to represent a single decimal digit. Binary numbers in the BCD format do not exceed 9 (1001) maximum in a single digit. When adding or subtracting BCD numbers, we use special rules to keep all single digits within the range 0 to 9.

This format is used internally in calculators and digital clocks.

Below is a table showing BCD numbers and the decimal equivalent:

BCD patterns and equivalent decimal representation

2^3 2^2 2^1 2^0

8's 4's 2's 1's

A	B	C	D	decimal equivalent
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9

BCD is a convenient way to represent digits in a decimal number system and yet easily manipulate these numbers inside a digital circuitry.

We also need to represent characters that we type on a keyboard or see in a display. For many decades, the most common format for English language displays is ASCII (American Standard Code for Information Interchange). This is an 8-bit code and fits conveniently within a single byte of 8 bits. All of the alphabetic characters, small size and capitalized letters, number, punctuation marks, and a variety of special control codes can be encoded in this system. There is an ASCII table in your textbook on page 27, table 1-5 that shows the encoding. From this table, we will see that the capital letter 'A' is encoded as: 1000001. This is hexadecimal 0x41. Note that the small letter 'a' is encoded as: 1100001 or hexadecimal 0xC1. The capital letter 'T' for example is encoded 1010100. You may have noted there are only 7 bits shown in the table and in the above examples. All of the ASCII characters can in fact be encoded in 7 bits (128 total combinations) as shown in the table. However, computers internally store ASCII characters in 8-bit bytes for convenience. The 8th bit, the MSbit can be used as a parity bit or a special purpose bit. When not used for a special purpose, the MSbit is normally set to 0.

Parity bits are not used as often these days. Parity bits was a method in common use a few decades ago to help with detecting data errors. Basically a parity bit could be either odd or even. If using odd parity, the number of ones in data (say 7 bits of an ASCII character) are determined. The parity bit is then set as appropriate so the total number of ones including the parity bit would be an odd number. A similar approach used for even parity. On the receiving end of a communications system, the parity bit would be calculated and compared. If parity bit did not match, then the data was assumed to be

corrupted and the data re-sent. Parity can detect single bit errors, but may miss double bit errors. In modern times, we normally use more sophisticated methods such as checksum on packet of data, or CRC checks on large files of data. Data transmission is generally reliable these days, so use of the parity bit is not common now. It is an inefficient method since it requires an extra 'bit' of overhead in the data transmission.

We will stop here for today. Next time we will review the gray code and then move on to the material in chapter 2 on combinational logic circuits.

We will have some homework assignments and some short quizzes along the way as well as an exam about once per week. The first exam will be toward the end of next week and is on the schedule on the syllabus. All exams and quizzes are open notes and open book, any calculator, entirely individual work.