Fundamentals of Applied Microcontrollers Manual

Seth McNeill

Edition 0.01
2022 January 20

Contents

Chapter 1

Introduction

This book is the accompanying text to a class introducing microcontrollers to upper division, non-electrical engineering undergraduate students who have taken some C programming.

Chapter 2

Number Systems

This chapter introduces the concepts of different numbering systems that are relevant to microcontrollers.

2.1 Decimal Numbers

The standard number system used by humans is based on 10. This logically flows the usual numbers of fingers or toes humans have. When counting in the base 10 (decimal) we start at 0, count up to 9, then run out of numbers to use. When runs out of numbers, we put a number to the left of the column we were counting in, and then start over at zero again which gives us the number 10. A more clear way to count would be to count from 00 to 09, then increment the left digit and start the right digit back at 0. This can be continued until 99 is reached. But now we have a model to follow. If a column gets to 9, we increment a column to the left and start the current column over at 0. This leads to the number 100. This concept can be continued forever.

When given a particular number, 3254, in decimal the value is calculated as

$$3254 = 3 \cdot 1000 + 2 \cdot 100 + 5 \cdot 10 + 4 \tag{2.1}$$

It could also be represented as

$$3254 = 3 \cdot 10^3 + 2 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0 \tag{2.2}$$

This is the basis for a positional number system.

2.2 Binary Numbers

Computers run a base 2 system, also known as binary. This means that we only have two options at each position, 1 or 0, instead of the 10 available in the decimal system. However, counting is done in the same way, when you run out of symbols, increment the position to the left and then start over at zero in the current position.

Binary numbers are also positional so a number like 1011 is

$$1011 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11_{10} \tag{2.3}$$

where 11_{10} indicates that the number 11 is base 10.

The maximum value a binary number can have is determined by the number of bits it has. The formula is:

$$V_{max} = 2^n - 1 \tag{2.4}$$

where n is the number of bits. For example, an 8 bit number has a maximum value of $2^8 - 1 = 255$.

2.3 Hexadecimal Numbers

The problem for humans is that we have a very difficult time reading binary numbers. Especially once the numbers get long such as 11101101010010110101_2. The first thing we can do to improve intelligibility is to put a space in every 4 digits so that we aren't completely bowled over by all the digits. This gives us 1110 1101 0100 1011 0101_2. This is better but still leaves some challenges. Someone pointed out that since 4 bits can have 16 different values, what if each 4 bits (also called a nibble) was represented by a single base 16 number. Base 16 is called hexadecimal. It has the problem that it runs out of digits once we get to 10, so it was decided to simply start on the alphabet so 10_{10} is A_{16}, 11_{10} is B_{16}, and so forth. Now the cumbersome binary number we've been playing with can be represented as $ED4B5_{16}$.

A comparison of the different numbering system representations of 0 through 15_{10} are shown in Table 2.1.

Decimal	Binary	Hexadecimal
00	0000	0
01	0001	1
02	0010	2
03	0011	3
04	0100	4
05	0101	5
06	0110	6
07	0111	7
08	1000	8
09	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Table 2.1: Decimal, binary, and hexadecimal numbers align as shown in this table.

2.4 Binary Background

The reasons for this stem from early computers using switches (relays) as their basic computing elements and the fact that telling the difference between a "high" voltage and a "low" voltage is easier to do than to differentiate 10 different voltages. It also allows for a gap between high and low that creates some immunity to noise. (TODO: insert figure or reference figure from logic chapter)

2.5 Converting Between Bases

2.5.1 Binary to Decimal

Converting to decimal is straightforward since we can simply sum each digit multiplied by the power of 2 it represents.

$$V_{10} = \sum_{i=0}^{p} d_i \cdot 2^i \tag{2.5}$$

In Equation 2.5 the p digits, d_i, of the binary number are summed from right to left while being multiplied by the power of 2 they represent. As an example, convert 1011 0111 to decimal.

$$\begin{aligned} V_{10} &= 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 + 1 \cdot 2^5 + 0 \cdot 2^6 + 1 \cdot 2^7 \\ &= 1 + 2 + 4 + 0 + 16 + 32 + 0 + 128 \\ &= 183 \end{aligned} \tag{2.6}$$

I usually find it easiest to just remember the powers of two for each place and add them up.

$$\begin{array}{cccccccc} {\scriptstyle 128} & {\scriptstyle 64} & {\scriptstyle 32} & {\scriptstyle 16} & {\scriptstyle 8} & {\scriptstyle 4} & {\scriptstyle 2} & {\scriptstyle 1} \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \end{array} = 1 + 2 + 4 + 16 + 32 + 128 \tag{2.7}$$

2.5.2 Decimal to Binary

Converting decimal numbers to binary involves dividing by 2 until the remainder is 0 or 1. The process goes as follows:

1. Divide the number by 2. If the remainder is 1 then the least significant bit is 1 otherwise if the remainder is 0 (it was an even number) then the least significant bit is 0.
2. Take the quotient from the previous step and divide by 2. The remainder is the next more significant bit in the binary representation.
3. Repeat Step 2 until the quotient is 0.

As an example, let's convert 11_{10} to binary.

$$\begin{aligned} 11 \div 2 &= 5R1 \quad \rightarrow 1 \text{ (LSB)} \\ 5 \div 2 &= 2R1 \quad \rightarrow 1 \\ 2 \div 2 &= 1R0 \quad \rightarrow 0 \\ 1 \div 2 &= 0R1 \quad \rightarrow 1 \text{ (MSB)} \end{aligned} \tag{2.8}$$

That means that 11_{10} is 1011_2. As another example let us convert 20_{10} to binary.

$$\begin{aligned} 20 \div 2 &= 10R0 \quad \rightarrow 0 \text{ (LSB)} \\ 10 \div 2 &= 5R0 \quad \rightarrow 0 \\ 5 \div 2 &= 2R1 \quad \rightarrow 1 \\ 2 \div 2 &= 1R0 \quad \rightarrow 0 \\ 1 \div 2 &= 0R1 \quad \rightarrow 1 \text{ (MSB)} \end{aligned} \tag{2.9}$$

This shows that 20_{10} is 10100_2.

2.5.3 Binary and Hexadecimal

For conversions between binary and hexadecimal I tend to use the table lookup method. After using it enough times you begin to memorize the conversions. In my head I'm usually converting from binary to decimal on each nibble and then converting decimal into hexadecimal. So if I see 0101 I remember it is 5 in decimal which is the same in hex. If I see 1010 I remember that it is $8 + 2 = 10$ which is one more than 9 so it is A in hex.

2.6 Colors

Colors for display on computers are represented as binary numbers. Colors are 24 bit which is broken down into 3 8 bit numbers representing red, green, and blue (RGB). Eight bits give values over the range of 0 to 255 so colors are represented by 3 numbers such as (255, 0, 255) which gives a color like this. It is good to note that (255, 255, 255) is white, (0, 0, 0) is black, and (X, X, X) where all three numbers are the same is gray.

Sometimes colors are represented as a 6 digit binary number in hexadecimal form prefixed by 0x such as 0xFF00FF for the color we looked at previously.

2.7 ASCII

Since computers operate using binary numbers, how do we get them to represent human languages? The method caught it is called American Standard Code for Information Interchange (ASCII). Basically, 7 bit numbers were

mapped to letters, numbers, punctuation, symbols, and control characters. Some examples are A is 65, Z is 90, a is 97, z is 122, 0 is 48, 9 is 57. The most used control characters are carriage return (13 or \r) and line feed (10 or \n). Unix based operating systems (Linux, OS X) use line feed to indicate a new line. Windows uses both (\r\n).

Look up an ASCII table online when you need to know the values.

2.8 Adding and Subtracting Binary Numbers

2.8.1 Default method

2.8.1.1 Addition

Calculate the binary sum 0001 0011 1101 + 0000 1011 0111.

$$\begin{array}{rcccccccccccc} & & & & & & \overset{1}{0} & \overset{1}{1} & \overset{1}{1} & \overset{1}{1} & \overset{1}{1} & \overset{1}{0} & \\ & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\ + & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ \hline & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \end{array} \tag{2.10}$$

2.8.1.2 Subtraction

Calculate the binary difference 0001 0011 1101 − 0000 1011 0111.

$$\begin{array}{rcccccccccccc} & & & & & \overset{0}{\not 1} & \overset{01}{\not 0} & & & \overset{0}{\not 1} & \overset{01}{\not 1} & \overset{01}{\not 0} & \\ & 0 & 0 & 0 & \not 1 & \not 0 & 0 & 1 & 1 & \not 1 & \not 1 & \not 0 & 1 \\ - & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ \hline & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \tag{2.11}$$

2.8.2 Twos Complement method

2.9 Gray Codes

When counting in binary, often times more than one bit changes as the number increments. This works fine in the ideal world, but in the real world, the logic controlling each bit might be slightly different. This will cause

Decimal	Two's Complement
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
−1	1111
−2	1110
−3	1101
−4	1100
−5	1011
−6	1010
−7	1001
−8	1000

Table 2.2: This table shows the decimal and two's complement numbers for 4-bits.

one bit to change at a slightly different time than another bit. That causes glitches in the counting that can be disruptive to the overall system.

Take as an example a robot that represents the cardinal directions as binary integers as illustrated in Figure 2.1.

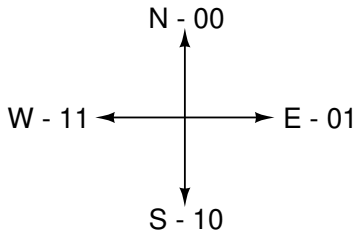

Figure 2.1: This figure shows using regular binary counting to represent the cardinal directions.

Using regular binary counting means that as the direction changes from East to South, two bits have to change. If the bits are driven by real switches

or differing logic they may not change simultaneously leading to possible outputs of E - W - S or E - N - S. If the data is being used in a sequential manner it could lead to erroneous actions by the device (robot, car, etc.). Instead of using regular binary counting, we could use an encoding that only changes one bit at a time as the directions change as shown in Figure 2.2.

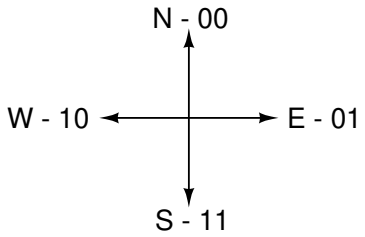

Figure 2.2: This figure shows using Gray codes to represent the cardinal directions.

Counting systems like this are called Gray code after Frank Gray or reflected binary code. A 4-bit example is shown in Table 2.3.

2.10 Binary Background

Originally, computers were just a bunch of switches, therefore, they only had two positions: on and off. This also has the benefit that there is a large amount of noise immunity. A repeater (buffer) can eliminate by recreating the original signal without noise. These are all great benefits of the binary system.

2.11 Assignment

There is a numbers quiz on Canvas as the assignment for this lab.

Gray Code
0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000

Table 2.3: This table shows 4-bit Gray codes with horizontal lines showing the break for 2- and 3-bit Gray codes.

Chapter 3

Boolean Logic

3.1 Introduction

This chapter introduces some basic Boolean logic including gates and Boolean algebra.

When can you drive through an intersection? When the light is NOT red. This is the first and simplest logic operator–the NOT element. It simply changes any TRUE to FALSE or FALSE to TRUE (you can substitute 1 for TRUE and 0 for FALSE, or on/off).

How do you start a car? In most of the cars I have driven I have to press on the brake at the same time as I turn the key. To say it another way, the car starts when I press the break AND turn the key.

$$pressBreak \text{ AND } turnKey = startedCar \tag{3.1}$$

Again on a car, a particular blinker light will turn on if you turn on the turn signal or if you turn on the 4-way blinker.

$$turnSignal \text{ OR } 4wayBlinker = blinking \tag{3.2}$$

The AND and OR in the equation are Boolean operators.

3.2 Methods of Representing Logic

It is important to differentiate between different forms of representation because $\overline{\text{EN}}$ means **active low** enable. It does not mean NOT(E AND N). The

understanding of which method is being used is usually derived from context. Using the dot ($\cdot$) everywhere can become burdensome, so when context makes it obvious (usually examples involving A, B, and C or X and Y) we may drop the dots and just use adjacency to represent the AND function.

3.3 Boolean Algebra

3.3.1 Theorems of Boolean Algebra

Ways to show NOT:

$$NOT(X) = \overline{X} = X' \tag{3.3a}$$

$$\overline{(\overline{X})} = (X')' = X \tag{3.3b}$$

Rules of AND and OR:

AND	OR	
$0 \cdot 0 = 0$	$1 + 1 = 1$	(3.4a)
$1 \cdot 1 = 1$	$0 + 0 = 0$	(3.4b)
$0 \cdot 1 = 1 \cdot 0 = 0$	$1 + 0 = 0 + 1 = 1$	(3.4c)

AND	OR	
$X \cdot 1 = X$	$X + 0 = X$	(3.5a)
$X \cdot 0 = 0$	$X + 1 = 1$	(3.5b)
$X \cdot X = X$	$X + X = X$	(3.5c)
$X \cdot \overline{X} = X$	$X + \overline{X} = 1$	(3.5d)
		(3.5e)

3.4 Logic Gates

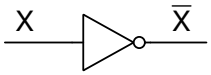

Figure 3.1: The NOT gate, also called an inverter, outputs NOT(A).

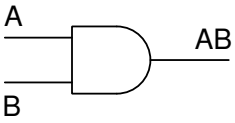

Figure 3.2: The AND gate outputs A AND B.

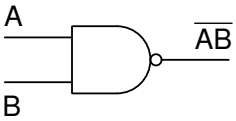

Figure 3.3: The NAND gate outputs NOT(A AND B).

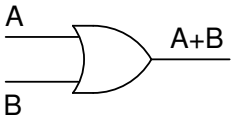

Figure 3.4: The OR gate outputs A OR B.

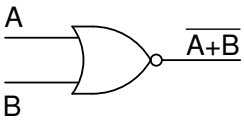

Figure 3.5: The NOR gate outputs NOT(A OR B).