## What you will learn

In this chapter you will learn:

1. binary numbers, and how they relate to computer hardware.
2. to convert to/from binary, decimal, and hexadecimal
3. binary character data representation in ASCII
4. integer numbers, which are represented in binary 2's complement format.
5. arithmetic operations for binary numbers
6. binary logic operations
7. the effect of context on data values in a computer.

# Chapter 1 Introduction

One of the major goals of computer science is to use abstraction to insulate the users from how the computer works. For instance, computers can interpret speech and use natural language processing to allow novice users to perform some amazing tasks. Even programming languages are written to enhance the ability of the person writing the code to create and support the program, and a goal of most modern languages is to be hardware agnostic.

Abstraction is a very positive goal, but at some level a computer is just a machine. While High Level Languages (HLL) abstract and hide the underlying hardware, they must be translated into assembly language to use the hardware. One of the goals of a computer science education is to strip away this abstraction and make the workings of the computing machine clear. Without an understanding of a computer as a machine, even the best programmer, system administrator, support staff, etc., will have significant gaps in what they are able to accomplish. A basic understanding of hardware is important to any computer professional.

Learning assembly language is different than learning an HLL. Assembly is intended to directly manipulate the hardware that a program is run on. It does not rely on the ability to abstract behavior, instead it gives the ability to specify exactly how the hardware is to work. Therefore, it uses a very different vocabulary than an HLL. That vocabulary is not composed of statements, variables, and numbers but of operations, instructions, addresses, and bits.

In assembly it is important to remember that the actual hardware to be used only understands binary values 0 and 1. To begin studying assembly, the reader must understand the basics of binary and how it is used in assembly language programming. The chapter is written to help the reader with the concepts of binary numbers.

## Chapter 1. 1 Binary Numbers

### Chapter 1.1. 1 Values for Binary Numbers

Many students will have had a class covering logic or Boolean algebra, where the binary values are generally true(T) and false(F) and use special symbols such as "^" for AND and "∨" for OR. This might be fine for mathematics and logic but is hopelessly inadequate for the engineering task of creating computer machines and languages.

To begin, the physical implementation of a binary value in a Central Processing Unit's (CPU) hardware, called a bit, is implemented as a circuit called a *flip-flop* or *latch.* A flip-flop has a voltage of either 0 volts or a positive voltage (most computers use +5 volts, but many modern computers use +3.3 volts, and any positive voltage is valid). If a flip-flop has a positive voltage, it is called *high* or *on* (true), and if it has 0 volts it is *low* or *off* (false). In addition, hardware is made up of gates that which can either be *open* (true) or *closed* (false). Finally, the goal of a computer is to be able to work with data that a person can understand. These numbers are always large, and hard to represent as a series of true or false values. When the values become large, people work best with numbers, so the binary number 0 is called false, and 1 is called true. Thus, while computers work with binary, there are several ways we can talk about that binary. If the discussion is about memory, the value is *high*, *on*, or *1*. When the purpose is to describe a gate, it is *open/closed*. If there is a logical operations values can be *true/false*. The following table summarizes the binary value naming conventions.

| T/F | Number | Switch | Voltage | Gate |
|---|---|---|---|---|
| F | 0 | Off | Low | Closed |
| T | 1 | On | High | Open |

**Table 1-1: Various names for binary values**

In addition to the various names, engineers are more comfortable with real operators. This book will follow the convention that "+" is an OR operator, "*" is an AND operator, and "!" (pronounced *bang*) is a not operator.

Some students are uncomfortable with the ambiguity in the names for true and false. They often feel that the way the binary values were presented in their mathematics classes (as true/false) is the "correct" way to represent them. But keep in mind that this class is about implementing a computer in hardware. There is no correct, or even more correct, way to discuss binary values. How they will be referred to will depend on the way in which the value is being used. Understanding a computer requires the individual to be adaptable to all these ways of referring to binary values. They will all be used in this text, though most of the time the binary values of 0 and 1 will be used.

## Chapter 1.1. 2 Binary Whole Numbers

The numbering system that everyone learns in school is called *decimal* or *base 10*. This numbering system is called decimal because it has 10 digits, [0...9]. Thus, quantities up to 9 can be easily referenced in this system by a single number.

Computers use switches that can be either on (1) or off (0), and so computers use the binary, or base 2, numbering system. In binary, there are only two digits, 0 and 1, so values up to 1 can be easily represented by a single digit. Having only the ability to represent 0 or 1 items is too limiting to be useful. But then so are the 10 values which can be used in the decimal system. The question is how does the decimal handle the problem of numbers greater than 9, and can binary use the same idea?

In decimal when 1 is added to 9 the number 10 is created. The number 10 means that there is 1 group of ten numbers, and 0 one number. The number 11 is 1 group of 10 and 1 group of one. When 99 is reached, we have 100, which is 1 group of hundred, 0 tens, and 0 ones. So, the number 1,245 would be:

$$1{,}245 = 1*10^3 + 2*10^2 + 4*10^1 + 5*10^0$$

Base 2 can be handled in the same manner. The number $10_2$ (base 2) is 1 group of two and 0 ones, or just $2_{10}$ (base 10).[1] Counting in base 2 is the same. To count in base 2, the numbers are $0_2$, $1_2$, $10_2$, $11_2$, $100_2$, $101_2$, $110_2$ $111_2$, etc. Any number in base 2 can be converted to base 10 using this principal. Consider $101011_2$, which is:

$$1*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 32 + 8 + 2 + 1 = 43_{10}$$

To work with base 2 number, it is necessary to know the value of the powers of 2. The following table gives the powers of 2 for the first 16 numbers (to $2^{15}$). It is highly recommended that students memorize at least the first 11 values of this table (to $2^{10}$), as these will be used frequently.

| **n** | **$2^n$** | | **n** | **$2^n$** | | **n** | **$2^n$** | | **n** | **$2^n$** |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | | 4 | 16 | | 8 | 256 | | 12 | 4096 |
| 1 | 2 | | 5 | 32 | | 9 | 512 | | 13 | 8192 |
| 2 | 4 | | 6 | 64 | | 10 | 1024 | | 14 | 16348 |
| 3 | 8 | | 7 | 128 | | 11 | 2048 | | 15 | 32768 |

**Table 1-2: Values of $2^n$ for n = 0...15**

The first 11 powers of 2 are the most important because the values of $2^n$ are named when n is a decimal number evenly dividable by 10. For example, $2^{10}$ is 1 Kilo, $2^{20}$ is 1 Meg, etc. The name for these values of $2^n$ are given in the following table.

| | | | | | |
|---|---|---|---|---|---|
| $2^{10}$ | Kilo | $2^{30}$ | Giga | $2^{50}$ | Penta |
| $2^{20}$ | Mega | $2^{40}$ | Tera | $2^{60}$ | Exa |

**Table 1-3: Names for values of 2n, n = 10, 20, 30, 40, 50, 60**

Using these names and the values of $2^n$ from 0-9, it is possible to name all the binary numbers easily as illustrated below. To find the value of $2^{16}$, we would write:

$$2^{16} = 2^{10}*2^6 = 1K * 64 = 64K$$

[1] The old joke is that there are 10 types of people in the world, those who know binary and those who do not.

Older programmers will recognize this as the limit to the segment size on older PC's which could only address 16 bits. Younger students will recognize the value of $2^{32}$, which is:

$$2^{32} = 2^{30} * 2^2 = 1G * 4 = 4G$$

4G was the limit of memory available on more recent PCs with 32-bit addressing, though that limit has been moved with the advent of 64-bit computers.

## Chapter 1. 2 Converting Binary, Decimal, and Hex Numbers

### Chapter 1.2. 1 Converting Binary to Decimal

Computers think in 0's and 1's, and when dealing with the internal workings of a computer humans must adjust to the computer's mindset. However, when the computer produces answers, the humans that use them like to think in decimal. So, it is often necessary for programmers to be able to convert between what the computer wants to see (binary) and what the human end users want to see (decimal). These next 3 sections will deal with how to convert binary to decimal, and then give 2 ways to convert decimal to binary. Finally, it will give a section on a useful representation for handling large binary numbers called hexadecimal.

To convert binary to decimal, it is only necessary to remember that each 0 or 1 in a binary number represents the amount of that binary power of 2. For each binary power of 2, you have either 0 or 1 instance of that number. To see this, consider the binary number $1001010_2$. This number has $1 * 2^6 + 0 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 64 + 8 + 2 = 74_{10}$. This can be generalized into an easy way to do this conversion. To convert from binary to decimal, put the $2^n$ value of each bit over the bits in the binary number and add the values which are 1, as in the example below:

$$\begin{array}{rccccccccl} & 64 & 32 & 16 & 8 & 4 & 2 & 1 & \\ 1001010_2 = & 1 & 0 & 0 & 1 & 0 & 1 & 0 & = 64 + 8 + 2 = 74_{10} \end{array}$$

### Chapter 1.2. 2 Converting Decimal to Binary using Binary Powers

Two ways to convert decimal number to binary numbers are presented here. The first is easy to explain, but harder to implement. The second is a cleaner algorithm, but why the algorithm works is less intuitive.

The first way to convert a number from decimal to binary is to see if a power of 2 is present in the number. For example, consider the number 433. We know that there is no $2^9$ (512) value in 433, but there is one value of $2^8$ (or 256). So, in the 9th digit of the base 2 number, we would put a 1, and subtract that 256 from the value of 433.

433 - 256 = 177

| $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|
| 1 | - | - | - | - | - | - | - | - |

$$\left.\begin{array}{cccc} 1 & 0 & 1 & 0 \\ 2^3 & 2^2 & 2^1 & 2^0 \end{array}\right\} 2^3 + 0 + 2 + 0 = 8 + 2 = 10 \text{ decimal}$$

$$\left.\begin{array}{ccccc} 1 & 0 & 0 & 1 & 0 \\ 2^4 & 2^3 & 2^2 & 2^1 & 2^0 \end{array}\right\} 2^4 + 0 + 0 + 2 + 0 = 18 \text{ decimal}$$

Next check if there is a $2^7$ (128) value in the number. There is, so add that bit to our string and subtract 128 from the result.

177 - 128 = 49

| $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | - | - | - | - | - | - | - |

Now check for values of $2^6$ (64). Since 64 > 49, put a zero in the $2^6$ position and continue.

49 - 0 = 49

| $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | - | - | - | - | - | - |

Continuing this process for $2^5$ (32), $2^4$(16), $2^3$(8), $2^2$(4), $2^1$(2), and $2^0$(1) results in the final answer.

| $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |

Thus $433_{10} = 110110001_2$. This result can be checked by converting the base 2 number back to base 10.

## Chapter 1.2. 3 Converting Decimal to Binary using Division

While conceptually easy to understand, the method to convert decimal numbers to binary numbers in Chapter 1.2.2 is not easy to implement as the starting and stopping conditions are hard to define. There is a way to implement the conversion which results in a nicer algorithm.

The second way to convert a decimal number to binary is to do successive divisions by the number 2. This is because if a number is divided and the remainder taken, the remainder is the value of the $2^0$ bit. Likewise, if the result of step 1 is divided again by 2 (so essentially dividing by 2*2 or 4), the reminder is the value of the $2^1$ bit. This process is continued until the result of the division is 0. The example below shows how this works.

Start with the number 433. 433 divided by 2 is 216 with a remainder of 1. So, in step 1 the result would have the first bit for the power of 2 set to one, as below:

1

433 / 2 = 216 r 1

| $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | - | 1 |

The number 216 is now divided by 2 to give 108, and the remainder, zero, placed in the second bit.

2

216 / 2 = 108 r 0

| $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | 0 | 1 |

The process continues to divide by 2, filling the remainder in each appropriate bit, until the result is 0, as below.

3

| $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |

## Chapter 1.2. 4 Converting between binary and hexadecimal

One of the biggest problems with binary is that the numbers rapidly become very hard to read. This is also true in decimal, where there is often a "," inserted between groupings of $10^3$. So, for example 1632134 is often written as 1,632,134, which is easier to read.

In binary, something similar is done. Most students are familiar with the term byte, which is 8 bits. But fewer know of a nibble, or 4 bits. 4 bits in binary can represent numbers between 0...15, or 16 values. So, values of 4 bits are collected and create a base 16 number, called a hexadecimal (or simply hex) number. To do this, 16 digits are needed, and arbitrarily the numbers and letters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F were chosen as the 16 digits. The binary numbers corresponding to these 16 digit hex numbers are given in the table below (note, the normal way to indicate a value is in hex is to write a *0x* before it So decimal 10 would be *0xA)*.

| Binary Number | Hex Digit | Binary Number | Hex Digit | Binary Number | Hex Digit | Binary Number | Hex Digit |
|---|---|---|---|---|---|---|---|
| 0000 | 0x0 | 0001 | 0x 1 | 0010 | 0x 2 | 0011 | 0x 3 |
| 0100 | 0x 4 | 0101 | 0x 5 | 0110 | 0x 6 | 0111 | 0x 7 |
| 1000 | 0x 8 | 1001 | 0x 9 | 1010 | 0x A | 1011 | 0x B |
| 1100 | 0x C | 1101 | 0x D | 1110 | 0x E | 1111 | 0x F |

**Table 1-4: Binary to Hexadecimal Conversion**

The hex numbers can then be arranged in groups of 8 (or 32 bits) to make it easier to translate from a 32-bit computer.

Note that hex numbers are normally only used to represent groupings of 4 binary digits. Regardless of what the underlying binary values represent, hex will be used just to show what the binary digits are. So, in this text all hex values will be unsigned whole numbers.

Most students recognize that a decimal number can be extended by adding a 0 to the left of a decimal number, which does not in any way change that number. For example, $00433_{10} = 0433_{10} = 433_{10}$. The same rule applies to binary. So, the binary number $110110001_2 = 000110110001_2$.

But why would anyone want to add extra zeros to the left of a number? Because to print out the hex representation of a binary number, I need 4 binary digits to do it. The binary number 1 1011 $0001_2$ only has 1 binary digit in the high order byte. So, to convert this number to binary it is necessary to pad it with left zeros, which have no effect on the number. Thus 1 1011 $0001_2$ = 0001 1011 $0001_2$= 0x1B1 in hex. Note that even the hex numbers are often padded with zeros, as the hex number 0x1B1 is normally be written 0x01B1, to get groupings of 4 hex numbers (or 32 bits).

It is often the case where specific bits of a 32-bit number need to be set. This is most easily done using a hex number. For instance, if a number is required where all the bits except the right left most (or 1) bit of a number is set, you can write the number in binary as:

$$11111111111111111111111111111110_2$$

A second option is to write the decimal value as: $4294967295_{10}$

Finally, the hex value can be written as 0xFFFFFFFE

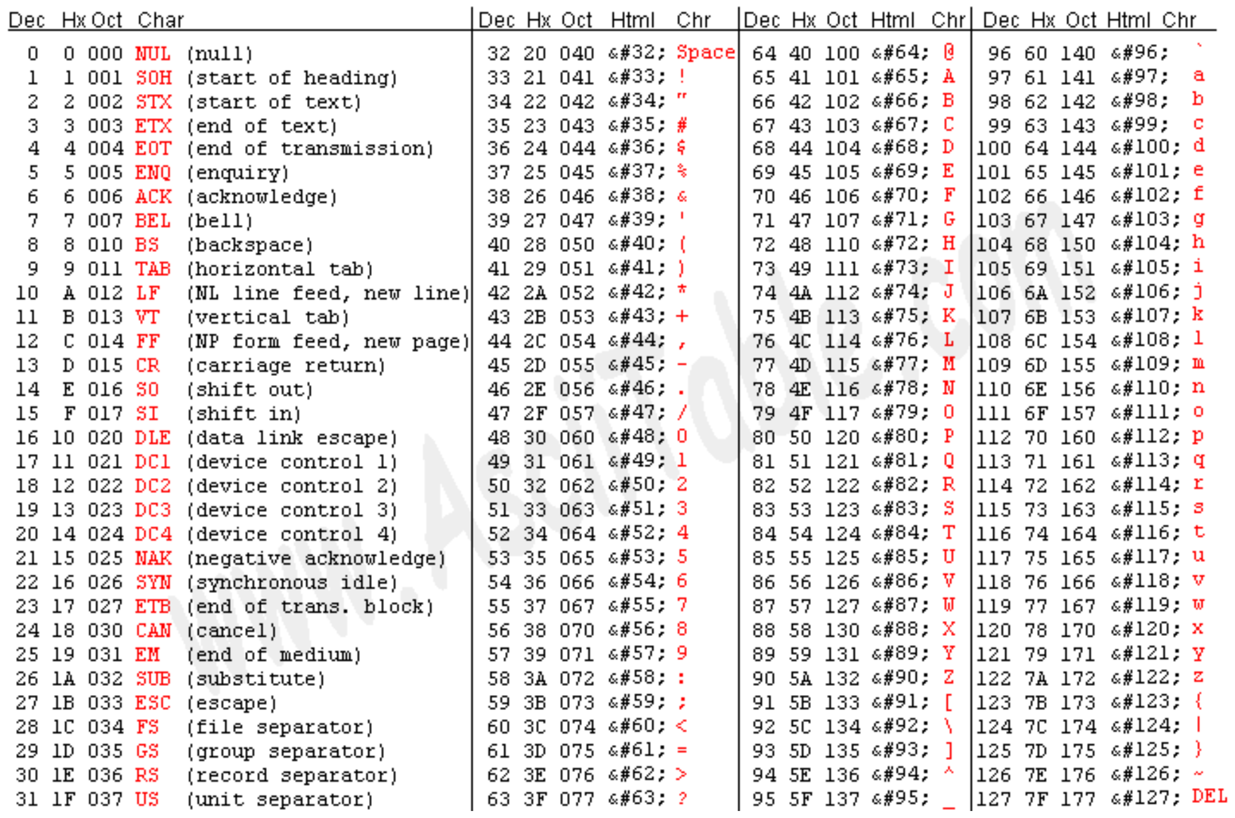
In almost all cases where specific bits are being set, a hex representation of the number is the easiest to understand and use.

## Chapter 1. 3 Character Representation

All the numbers used so far in this text have been binary whole numbers. While everything in a computer is binary, and can be represented as a binary value, binary whole numbers do not represent the universe of numbering systems that exists in computers. Two representations that will be covered in the next two sections are character data and integer data.

Though computers use binary to represent data, humans deal with information as symbolic alphabetic and numeric data. So, to allow computers to handle user readable alpha/numeric data, a system to encode characters as binary numbers was created. That system is called American Standard Code for Information Interchange (ASCII)[2]. In ASCII all characters are represented by a number from 1 - 127, stored in 8 bits. The ASCII encodings are shown in the following table.

[2] ASCII is limited to just 127 characters and is thus too limited for many applications that deal with internationalization using multiple languages and alphabets. Representations, such as Unicode, have been developed to handle these character sets, but are complex and not needed to understand MIPS Assembly. So, this text will limit all character representations to ASCII.



| Dec | Hx | Oct | Char | | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 000 | NUL | (null) | 32 | 20 | 040 | &#32; | Space | 64 | 40 | 100 | &#64; | @ | 96 | 60 | 140 | &#96; | ` |
| 1 | 1 | 001 | SOH | (start of heading) | 33 | 21 | 041 | &#33; | ! | 65 | 41 | 101 | &#65; | A | 97 | 61 | 141 | &#97; | a |
| 2 | 2 | 002 | STX | (start of text) | 34 | 22 | 042 | &#34; | " | 66 | 42 | 102 | &#66; | B | 98 | 62 | 142 | &#98; | b |
| 3 | 3 | 003 | ETX | (end of text) | 35 | 23 | 043 | &#35; | # | 67 | 43 | 103 | &#67; | C | 99 | 63 | 143 | &#99; | c |
| 4 | 4 | 004 | EOT | (end of transmission) | 36 | 24 | 044 | &#36; | $ | 68 | 44 | 104 | &#68; | D | 100 | 64 | 144 | &#100; | d |
| 5 | 5 | 005 | ENQ | (enquiry) | 37 | 25 | 045 | &#37; | % | 69 | 45 | 105 | &#69; | E | 101 | 65 | 145 | &#101; | e |
| 6 | 6 | 006 | ACK | (acknowledge) | 38 | 26 | 046 | &#38; | & | 70 | 46 | 106 | &#70; | F | 102 | 66 | 146 | &#102; | f |
| 7 | 7 | 007 | BEL | (bell) | 39 | 27 | 047 | &#39; | ' | 71 | 47 | 107 | &#71; | G | 103 | 67 | 147 | &#103; | g |
| 8 | 8 | 010 | BS | (backspace) | 40 | 28 | 050 | &#40; | ( | 72 | 48 | 110 | &#72; | H | 104 | 68 | 150 | &#104; | h |
| 9 | 9 | 011 | TAB | (horizontal tab) | 41 | 29 | 051 | &#41; | ) | 73 | 49 | 111 | &#73; | I | 105 | 69 | 151 | &#105; | i |
| 10 | A | 012 | LF | (NL line feed, new line) | 42 | 2A | 052 | &#42; | * | 74 | 4A | 112 | &#74; | J | 106 | 6A | 152 | &#106; | j |
| 11 | B | 013 | VT | (vertical tab) | 43 | 2B | 053 | &#43; | + | 75 | 4B | 113 | &#75; | K | 107 | 6B | 153 | &#107; | k |
| 12 | C | 014 | FF | (NP form feed, new page) | 44 | 2C | 054 | &#44; | , | 76 | 4C | 114 | &#76; | L | 108 | 6C | 154 | &#108; | l |
| 13 | D | 015 | CR | (carriage return) | 45 | 2D | 055 | &#45; | - | 77 | 4D | 115 | &#77; | M | 109 | 6D | 155 | &#109; | m |
| 14 | E | 016 | SO | (shift out) | 46 | 2E | 056 | &#46; | . | 78 | 4E | 116 | &#78; | N | 110 | 6E | 156 | &#110; | n |
| 15 | F | 017 | SI | (shift in) | 47 | 2F | 057 | &#47; | / | 79 | 4F | 117 | &#79; | O | 111 | 6F | 157 | &#111; | o |
| 16 | 10 | 020 | DLE | (data link escape) | 48 | 30 | 060 | &#48; | 0 | 80 | 50 | 120 | &#80; | P | 112 | 70 | 160 | &#112; | p |
| 17 | 11 | 021 | DC1 | (device control 1) | 49 | 31 | 061 | &#49; | 1 | 81 | 51 | 121 | &#81; | Q | 113 | 71 | 161 | &#113; | q |
| 18 | 12 | 022 | DC2 | (device control 2) | 50 | 32 | 062 | &#50; | 2 | 82 | 52 | 122 | &#82; | R | 114 | 72 | 162 | &#114; | r |
| 19 | 13 | 023 | DC3 | (device control 3) | 51 | 33 | 063 | &#51; | 3 | 83 | 53 | 123 | &#83; | S | 115 | 73 | 163 | &#115; | s |
| 20 | 14 | 024 | DC4 | (device control 4) | 52 | 34 | 064 | &#52; | 4 | 84 | 54 | 124 | &#84; | T | 116 | 74 | 164 | &#116; | t |
| 21 | 15 | 025 | NAK | (negative acknowledge) | 53 | 35 | 065 | &#53; | 5 | 85 | 55 | 125 | &#85; | U | 117 | 75 | 165 | &#117; | u |
| 22 | 16 | 026 | SYN | (synchronous idle) | 54 | 36 | 066 | &#54; | 6 | 86 | 56 | 126 | &#86; | V | 118 | 76 | 166 | &#118; | v |
| 23 | 17 | 027 | ETB | (end of trans. block) | 55 | 37 | 067 | &#55; | 7 | 87 | 57 | 127 | &#87; | W | 119 | 77 | 167 | &#119; | w |
| 24 | 18 | 030 | CAN | (cancel) | 56 | 38 | 070 | &#56; | 8 | 88 | 58 | 130 | &#88; | X | 120 | 78 | 170 | &#120; | x |
| 25 | 19 | 031 | EM | (end of medium) | 57 | 39 | 071 | &#57; | 9 | 89 | 59 | 131 | &#89; | Y | 121 | 79 | 171 | &#121; | y |
| 26 | 1A | 032 | SUB | (substitute) | 58 | 3A | 072 | &#58; | : | 90 | 5A | 132 | &#90; | Z | 122 | 7A | 172 | &#122; | z |
| 27 | 1B | 033 | ESC | (escape) | 59 | 3B | 073 | &#59; | ; | 91 | 5B | 133 | &#91; | [ | 123 | 7B | 173 | &#123; | { |
| 28 | 1C | 034 | FS | (file separator) | 60 | 3C | 074 | &#60; | < | 92 | 5C | 134 | &#92; | \ | 124 | 7C | 174 | &#124; | \| |
| 29 | 1D | 035 | GS | (group separator) | 61 | 3D | 075 | &#61; | = | 93 | 5D | 135 | &#93; | ] | 125 | 7D | 175 | &#125; | } |
| 30 | 1E | 036 | RS | (record separator) | 62 | 3E | 076 | &#62; | > | 94 | 5E | 136 | &#94; | ^ | 126 | 7E | 176 | &#126; | ~ |
| 31 | 1F | 037 | US | (unit separator) | 63 | 3F | 077 | &#63; | ? | 95 | 5F | 137 | &#95; | _ | 127 | 7F | 177 | &#127; | DEL |

**Table 1-5: ASCII Table**

Using this table, it is possible to encode a string such as "Once" in ASCII characters as the hexadecimal number 0x4F6E6365[3] (capital O = 0x4F, n = 0x6E, c = 0x63, e = 0x65).

Numbers as character data are also represented in ASCII. Note the number 13 is 0xD or $1101_2$. However, the value of the character string "13" is 0x3133. When dealing with data, it is important to remember what the data represents. Character numbers are represented using binary values but are very different from their binary numbers.

Finally, some of the interesting patterns in ASCII should be noted. All digits start with binary digits 0011 0000. Thus 0 is 0011 0000, 1 is 0011 0001, etc. To convert a numeric character digit to a number it is only necessary to subtract the character value of 0. For example, '0' - '0' = 0, '1' - '0' = 1, etc. This is the basis for an easy algorithm to convert numeric strings to numbers which will be discussed in the problems.

Also note that all ASCII upper case letters start with the binary string 0010 0001 and are 1 offset for each new character. So, *A* is 0100 0001, *B* is 0100 0010, etc. Lower case letters start with the binary string 0110 0001 and are offset by 1 for each new character, so *a* is 0110 0001, *b* is

[3] By now it is hoped that the reader is convinced that hexadecimal is a generally preferred way to represent data in a computer. The decimal value for this string would be 1,332,634,469 and the binary would be 0100 1111 0110 1110 0110 0011 0010 0101. Having taught for many years, however, I know old habits die hard with many students who will struggle endlessly converting to decimal to avoid learning hex.

0110 0010, etc. Therefore, all upper-case letters differ from their lower-case counterpart by a 1 in the digit 0100. This relationship between lower case and capital letters will be used to illustrate logical operations later in this chapter.

## Chapter 1. 4 Adding Binary Whole Numbers

Before moving on to how integer values are stored and used in a computer for calculations, how to do addition of binary whole numbers needs to be covered.

When 2 one-bit binary numbers are added, the following results are possible: $0_2+0_2 = 0_2$; $0_2+1_2 = 1_2$; $1_2+0_2 = 1_2$; and $1_2+1_2 = 10_2$. This is just like decimal numbers. For example, 3+4=7, and the result is still one digit. A problem occurs, however, when adding two decimal numbers where the result is greater than the base of the number (for decimal, the base is 10). For example, 9+8. The result cannot be represented in one digit, so a carry digit is created. The result of 9+8 is 7 with a carry of 1. The carry of 1 is considered the next digit, which is adding 3 digits (the two addends, and the carry). So, 39 + 28 = 67, where the 10's digit (6) is the result of the two addends (3 and 2) and the carry (1).

The result of $1_2+1_2 = 10_2$ in binary is analogous to the situation in base 10. The addition of $1_2+1_2$ is $0_2$ with a carry of $1_2$, and there is a carry to the next digit of $1_2$.
An illustration of binary addition is shown in the figure below.

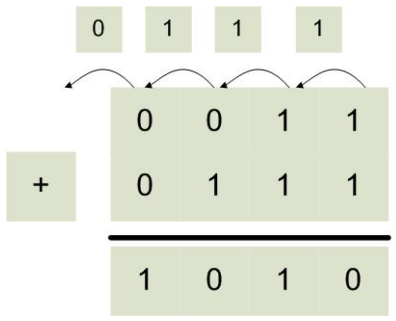


**Figure 1-1: Binary whole number addition**

Here the first bit adds $1_2 +1_2$, which yields a $0_2$ in this bit and a carry bit of $1_2$. The next bit now must add $1_2 +1_2 +1_2$ (the extra one is the carry bit), which yields a $1_2$ for this bit and a carry bit of $1_2$. If you follow the arithmetic through, you have $0011_2$ ($3_{10}$) + $0111_2$ ($7_{10}$) = $1010_2$ ($10_{10}$).

## Chapter 1. 5 Integer Numbers (2's Complement)

### Chapter 1.5. 1 What is an Integer

Using only positive whole numbers is too limiting for any valid calculation, and so the concept of a binary negative number is needed. When negative values for the set of whole numbers are included with the set of whole numbers (which are positive), the resulting set is called *integer* numbers. Integers are non-fractional numbers which have positive and negative values.

When learning mathematics, negative numbers are represented using a sign magnitude format, where a number has a sign (positive or negative), and a magnitude (or value). For example, -3 is 3 units (it's magnitude) away from zero in the negative direction (it's sign). Likewise, +5 is 5 units away from zero in a positive direction. Signed magnitude numbers are used in computers, but not for integer values. For now, just realize that it is excessively complex to do arithmetic using signed magnitude numbers. There is a much simpler way to do things called 2's complement. This text will use the term integer and 2's complement number interchangeably.

## Chapter 1.5. 2 2's complement operation and 2's complement format

Many students get confused and somehow believe that a 2's complement has something to do with negative numbers, so this section will try to be as explicit here as possible. Realize that if someone asks, "What is a 2's complement?", they are asking two very different questions. There is a 2's complement *operation* which can be used to negate a number (e.g., translate 2 -> -2 or -5 -> 5). There is also a 2's complement *representation* (or format) of numbers which can be used to represent integers, and those integers can be positive and negative whole numbers.

To reiterate, the 2's complement *operation* can convert negative numbers to the corresponding positive values, or positive numbers to the corresponding negative values.

A 2's complement *representation* (or format) simply represents a number, either positive or negative. If you are ever asked if a 2's complement number is positive or negative, the only appropriate answer is yes, a 2's complement number can be positive or negative.

The following sections will explain how to do a 2's complement operation, and how to use 2's complement numbers. Being careful to understand the difference between a 2's complement operation and 2's complement number will be a big help to the reader.

## Chapter 1.5. 3 The 2's Complement Operation

A 2's complement operation is simply a way to calculate the negation of a 2's complement number. It is important to realize that creating a 2's complement operation (or negation) is not as simple as putting a minus sign in front of the number. A 2's complement operation requires two steps: 1 - Inverting all the bits in the number; and 2 - Adding $1_2$ to the number.

Consider the number $00101100_2$. The first step is to reverse all the bits in the number (which will be achieved with a bitwise ! operation. Note that the ! operator is a unary operation, it only takes one argument, not two.

$$!\,(00101100_2) = 11010011_2$$

Note that in the equation above the bits in the number have simply been reversed, with 0's becoming 1's, and 1's becoming 0's. This is also called a 1's complement, though in this text we will never use a 1's complement number.

The second step adds a $1_2$ to the number.

$$\begin{array}{r} 1101\ 0011_2 \\ \underline{0000\ 0001_2} \\ 1101\ 0100_2. \end{array}$$

Thus, the result of a 2's complement operation on $00101100_2$ is $11010100_2$, or negative 2's complement value. This process is reversible, as the reader can easily show that the 2's complement value of $11010100_2$ is $00101100_2$. Also note that both the negative and positive values are in 2's complement representation.

While positive numbers will begin with a 0 in the left most position, and negative numbers will begin with a 1 in the leftmost position, these are not just sign bits in the same sense as the signed magnitude number, but part of the representation of the number. To understand this difference, consider the case where the positive and negative numbers used above are to be represented in 16 bits, not 8 bits. The first number, which is positive, will extend the sign of the number, which is 0. As we all know, adding 0's to the left of a positive number does not change the number. So, $00101100_2$ would become $0000000000101100_2$.

However, the negative value cannot extend 0 to the left. If for no other reason, this results in a 0 in the sign bit, and the negative number has been made positive. So, to extend the negative number $11010100_2$ to 16 bits requires that the sign bit, in this case 1, be extended. Thus $11010100_2$ becomes $1111111111010100_2$.

The left most (or high) bit in the number is normally referred to as a sign bit, a convention this text will continue. But it is important to remember it is not a single bit that determines the sign of the number, but a part of the 2's complement representation.

## Chapter 1.5. 4 The 2's Complement (or Integer) Type

Because the 2's complement operation negates a number, many people believe that a 2's complement number is negative. A better way to think about a 2's complement number is that is a type. A type is an abstraction which has a range of values, and a set of operations. For a 2's complement type, the range of values is all positive and negative whole numbers. For operations, it has the normal arithmetic operations such as addition (+), subtraction (-), multiplication (*), and division (/).

A type also needs an internal representation. In mathematics classes, numbers were always abstract, theoretical entities, and assumed to be infinite. But a computer is not an abstract entity, it is a physical implementation of a computing machine. Therefore, all numbers in a computer must have a physical size for their internal representation. For integers, this size is often 8 (byte), 16(short), 32(integer), or 64(long) bits, though larger numbers of bits can be used to store the numbers. Because the left most bit must be a 0 for positive, and 1 for negative, using a fixed size also helps to identify easily if the number is positive or negative.

Because the range of the integer values is constrained to the $2^n$ values (where n is the size of the integer) that can be represented with 2's complement, about half of which are positive, and half are negative, roughly $2^{n-1}$ values of magnitude are possible. However, one value, zero, must be accounted for, so there are 1 less positive number than negative numbers. So, while $2^8$ is 256, the 2's complement value of an 8-bit number runs from -128 ... 127.

Finally, as stated in the previous section, just like zeros can be added to the left of a positive number without effecting its value, in 2's complement, ones can be added to the left of a negative number without effecting its value. For example:

$$0010_2 = 0000\ 0010_2 = 2_{10}$$

$$1110_2 = 1111\ 1110_2 = -2_{10}$$

Adding leading zeros to a positive number, and leading ones to a negative number, is called sign extension of a 2's complement number.

## Chapter 1. 6 Integer Arithmetic

### Chapter 1.6. 1 Integer Addition

Binary whole number addition was covered in chapter 1.4. Integer addition is like binary whole number addition except that both positive and negative numbers must be considered. For example, consider adding the two positive numbers $0010_2$ ($2_{10}$) + $0011_2$ ($3_{10}$) = $0101_2$ ($5_{10}$).

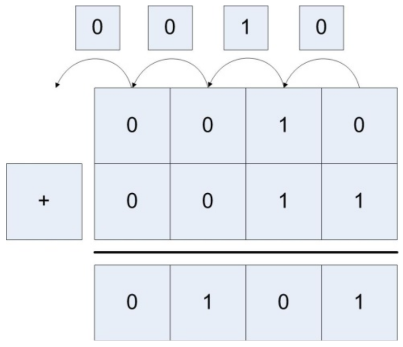


**Figure 1-2: Addition of two positive integers**

Addition of mixed positive and negative numbers, and two negative numbers also works in the same manner, as the following two examples show. Below we first show adding $0010_2$ ($2_{10}$) + $1101_2$ ($-3_{10}$) = $1111_2$ ($-1_{10}$), and the second we add $1110_2$ ($-2_{10}$) + $1101_2$ ($-3_{10}$) = $1011_2$ ($-5_{10}$).

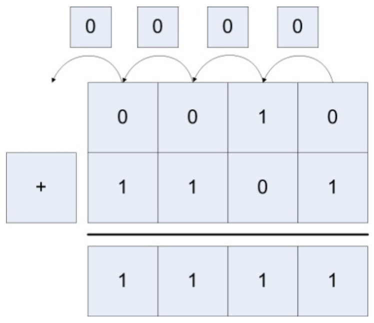


**Figure 1-3: Addition of positive and negative integers**

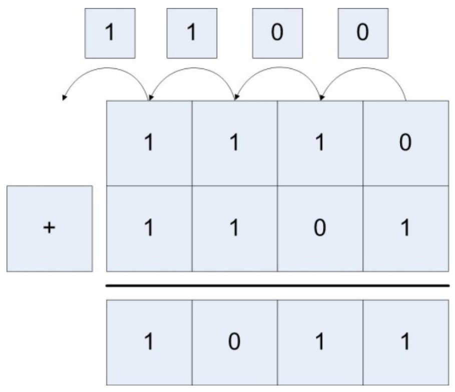


**Figure 1-4: Addition of two negative integers**

## Chapter 1.6. 2 Overflow of Integer Addition

Because integers have fixed sizes, addition and subtraction can cause a problem known as integer overflow. This happens when the two numbers being added are large positive or negative values, and the combining of the values results in numbers too big to be stored in the integer value.

For example, a 4-bit integer can store values from -8...7. So, when $0100_2$ ($4_{10}$) + $0101_2$ ($5_{10}$) = $1001_2$ ($-7_{10}$) are added using 4-bit integers the result is too large to store in the integer. When this happens, the number changes sign and gives the wrong answer, as the following figure shows.

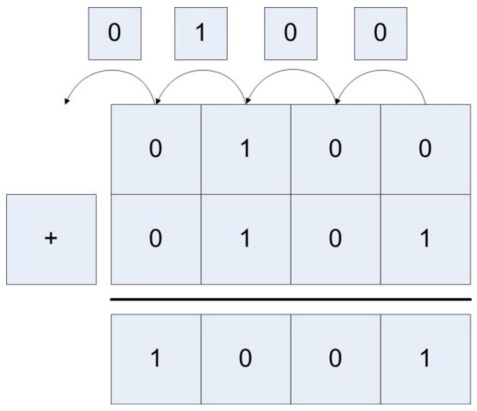


**Figure 1-5: Addition with overflow**

Attempting to algorithmically figure out if overflow occurs is difficult. First if one number is positive and the other is negative, overflow never occurs. If both numbers are positive or negative, then if the sign of the sum is different than the sign of either of the inputs, overflow has occurred.

There is a much easier way to figure out if overflow has occurred. If the carry in bit to the last digit is the same as the carry out bit, then no overflow has occurred. If they are different, then overflow has occurred. In Figure 1.3 the carry in and carry out for the last bit are both 0, so there is no overflow. Likewise, in Figure 1.4 the carry in and carry out are both 1, so there was no overflow. In Figure 1.5 the carry in is 1 and the carry out is 0, so overflow has occurred.

This method also works for addition of negative numbers. Consider adding $1100_2$ ($-4_{10}$) and $1011_2$ ($-5_{10}$) = $0111_2$ ($7_{10}$), shown in Figure 1.6. Here the carry in is 0 and the carry out is 1, so once again overflow has occurred.

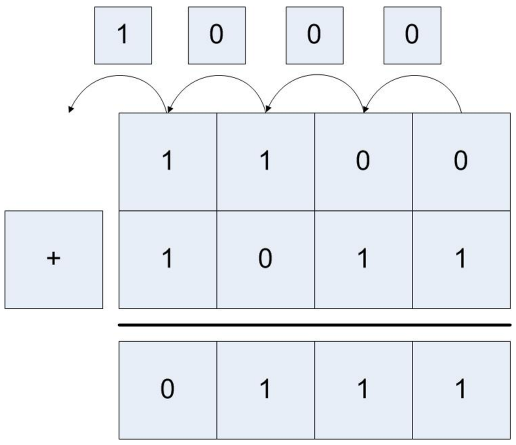


**Figure 1-6: Subtraction with overflow**

## Chapter 1.6. 3 Integer multiplication using bit shift operations

Multiplication and division of data values or variables involves hardware components in the Arithmetic Logic Unit (ALU). In assembly, these operations will be provided by various operators (`mul` and `div`), and the hardware to implement them is beyond the scope of this book and will not be covered. However, what is of interest in writing assembly is multiplication and division by a constant.

The reason multiplication and division by a constant is covered is that these operations can be provided by bit shift operations, and the bit shift operations are often faster to run than the equivalent `mul` or `div` operation. Therefore, bit shift operations are often used in assembly to do multiplication and division, and therefore it is important for assembly language programmers to understand how this works.

First consider multiplication of a number by a power of 10 in base 10. In base 10, if a number is multiplied by a power of 10 ($10^n$, where n is the power of 10), it is sufficient to move the number n places to the left filling in with 0's on the right. For example, 15*1000 (or 15 * $10^3$) = 15,000.

This same concept holds in binary. To multiply a binary number (e.g., 15, or $0000\ 1111_2$) by 2, the number is shifted to the left 1 digit (written as 1111<<1), yielding $0001\ 1110_2$ or 30. Likewise multiplying $0000\ 1111_2$ by 8 is done by moving the number 3 spaces to the left

($0000\ 1111_2$<<3), yielding $0111\ 1000_2$, or 120. So, it is easy to multiply any number represented in base 2 by a power of 2 (for example $2^n$) by doing n left bit shifts and backfilling with 0's.

Note that this also works for multiplication of negative 2's complement (or integer) numbers. Multiplying $1111\ 0001_2$ ($-15_{10}$) by 2 is done by moving the bits left 1 space and again appending a 0, yielding $1110\ 0010_2$ (or $-30_{10}$) (note that in this case 0 is used for positive or negative numbers). Again multiply $1111\ 0001_2$ ($-15_{10}$) by 8 is done using 3 bit-shifts and backfilling the number again with zeros, yielding $1000\ 1000_2$ ($-120_{10}$)

By applying simple arithmetic, it is easy to see how to do multiplication by a constant 10. Multiplication by 10 can be thought of as multiplication by (8+2), so (n*10) = ((n*8) + (n*2)).

$$15*10 = 15 * (8+2) = 15 *8 + 15 * 2 = (0000\ 1111_2 << 3) + (0000\ 1111_2 << 1) =$$

$$0111\ 1000_2 + 0001\ 1110_2 = 1001\ 0110_2 = 150$$

This factoring procedure applies for multiplication by any constant, as any constant can be represented by adding powers of 2. Thus, any constant multiplication can be encoded in assembly as a series of shifts and adds. This is sometimes faster, and often easier, than doing the math operations, and should be something every assembly language programmer should be familiar with.

This explanation of the constant multiplication trick works in assembly, which begs the question does it also work in a HLL? The answer is yes and no. Bit shifts and addition can be done in most programming languages, so constant multiplication can be implemented as bits shifts and addition. But because it can be done does not mean it should be done. In HLL (C/C++, Java, C#, etc.) this type of code is arcane, and difficult to read and understand. In addition, any decent compiler will convert constant multiplication into the correct underlying bit shifts and additions when it is more efficient to do so. And the compiler will make better decisions about when to use this method of multiplication and implement it more effectively and with fewer errors than if a programmer were to do it. So, unless there is some good reason to do multiplication using bit shifts and addition, it should be avoided in an HLL.

## Chapter 1.6. 4 Integer division using bit shift operations

Since multiplication can be implemented using bit shift operations, the obvious question is whether the same principal applies to division? The answer is that for some useful cases, division using bit shift operations does work. But in general, it is full of problems.

The cases where division using bit shift operations works are when the dividend is positive, and the divisor is a power of 2. For example, $0001\ 1001_2$ ($25_{10}$) divided by 2 would be a 1-bit shift, or $0000\ 1100_2$ ($12_{10}$). The answer 12.5 is truncated, as this is easily implemented by throwing away the bit which has been shifted out. Likewise, $0001\ 1001_2$ ($25_{10}$) divided by 8 is $0000\ 0011_2$ ($3_{10}$), with truncation again occurring. Also note that in this case the bit that is shifted in is the sign bit, which is necessary to maintain the correct sign of the number.

Bit shifting for division is useful in some algorithms such as a binary search to find parents in a c binary tree. But again, it should be avoided unless there is a strong reason to use it in a HLL. This leaves two issues. The first is why can this method not be implemented with constants other than the powers of 2. The reason is that division is only distributive in one direction over addition, and in our case, it is the wrong direction. Consider the equation 60/10. It is easy to show that division over addition does not work in this case.

$$60/10 = 60/(8+2) \neq 60/8 + 60/2$$

The second issue is why the dividend must be positive. To see why this is true, consider the following division, -15 / 2. This result in the following:

$$1111\ 0001_2 >> 1 = 1111\ 1000 = -8$$

Two things about this answer:
First in this case the sign bit, 1, must be shifted in to maintain the sign of the integer.

Second in this case the lowest bit, a 1, is truncated. This means that -7.5 is truncated down to -8. However, many programmers believe that -7.5 should truncate to -7. Whether the correct answer is -7 or -8 is debatable, and different programming languages have implemented as either value (for example, Java implements -15/2 = -7, but Python -15/2 as -8). This same problem occurs with many operations on negative numbers, such a modulus. And while such debates might be fun, and programmers should realize that these issues can occur, it is not the purpose of this book to do more than present the problem.

## Chapter 1. 7 Boolean Logical and Bitwise Operators

### Chapter 1.7. 1 Boolean Operators

Boolean operators are operators which are designed to operate on a Boolean or binary data. They take in one or more input values of 0/1[4] and combine those bits to create an output value which is either 0 or 1. This text will only deal with the most common Boolean operators, the unary operator NOT (or inverse), and the binary operators[5] AND, OR, NAND, NOR, and XOR. These operators are usually characterized by their truth tables, and two truth tables are given below for these operators.

| A | NOT |
|---|---|
| 0 | 1 |
| 1 | 0 |

**Table 1-6: Truth table for NOT operator**

[4] Note that the values 0/1 are used here rather than F/T. These operators will be described through the rest of the book using the binary values 0/1, so there is no reason to be inconsistent here.
[5] The term unary operator means having one input. The term binary operator means having two inputs. Be careful reading this sentence, as binary is used in two different contexts. The binary operator AND operates on binary data.

| Input | | Output | | | | |
|---|---|---|---|---|---|---|
| **A** | **B** | **AND** | **OR** | **NAND** | **NOR** | **XOR** |
| 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 |

**Table 1-7: Truth table for AND, OR, NAND, NOR, and XOR**

## Chapter 1.7. 2 Logical and Bitwise Boolean Operators

There are two kinds of Boolean operators implemented in many programming languages. They are logical operators and bitwise operators. Logical operators perform Boolean operations to obtain a single value at the end. For example, in Java a programmer might write:

```
        if ((x != 0) && (y / x > 4))
```

The purpose of this statement is to decide whether to enter the statement or code block associated with the if test. What is desired is a single answer, or a single bit, which is either true (1) or false (0). This has two consequences. The first is that in some programming languages (e.g., C/C++) a variable other than a Boolean can be used in the statement, with the consequence that 0 is false, and anything but 0 is true. For example, in the statement `if(x=64)`, the result is true. The equal operator returns a non-zero value, 64, which is true. This has been the cause of many bugs in C/C++, and most modern compilers at least complain about this. This will also be the result of some expressions in assembler, and there will be no compiler to complain about it. So be careful.

The second consequence of the logical Boolean operator is that once a part of it has failed, the entire statement must fail, and so the rest of the statement will not be executed. This is called short-circuiting, and all logical operators are thus short-circuiting operators. To see why this is true, consider the `if` test above. In this `if` test, if `x is 0`, then `(x != 0)` is false. Since *false* and *anything* is false, there is no need to evaluate the second part of this equation, and so the statement `(y / x > 4)` is not executed. Many programmers will recognize this code, as it is a common pattern for protecting against a divide by zero.

The important take away from this is that logical operators are short-circuiting operators.

On the other hand, bit-wise operators are not short-circuiting. Consider the following problem. A programmer wants to write a *toLower* method which will convert an upper-case letter to a lower-case letter. In Chapter 1.3 it was pointed out that the difference between an upper-case letter and a lower-case letter is that in a lower-case letter the bit 0x20 ($00100000_2$) is 1, whereas in the upper-case letter it is zero. So, to convert from upper-case to lower case, it is only necessary to OR the upper-case letter with 0x20. In pseudo code this could be implemented as follows:

```
        char toLower(char c) {
            return (c | 0x20)
        }
```

In this case the OR operator, |, needs to operate on every bit in the variables. Therefore, the | operator is not short-circuiting, it will process every bit regardless of whether a previous bit would cause the operation to fail.

It is possible to use bitwise operators in place of logical operators, but it is usually incorrect to do so. For example, in the previous if statement, if a bitwise operator had been used, no short-circuiting would have occurred and the divide by zero could occur.

```
        if ((x != 0) & (y / x > 4))
```

Many languages such as C/C++, Java, C#, etc., have both logical (short-circuiting) and bitwise operators. In most cases the single character operator is a bit wise operator (e.g., &, |, etc.) and the double character operator is the logical operator (&&, ||, etc.).

To make things more confusing, in MIPS (Microprocessor without Interlocked Pipeline Stages) only non-short-circuiting operators are used, however they are often called logical operators. There is no good way to reconcile this, so the user is cautioned to read the material and programs carefully.

## Chapter 1. 8 Context

The final bit of information to take from this chapter is that data in a computer is a series of "1" and "0" bits. In computer memory a byte containing “0100 0001” could exists. The question is what does this byte mean? Is the byte an integer number corresponding to decimal 65? Is this an ASCII character, representing the letter "A". Is it a floating-point number, or maybe an address? The answer is, you have no idea!

To understand data there must be a context. A HLL will always provide the context with the data (for example, the type, as in int a;), so the programmer does not have to worry about it. However, in assembly the only context is the one the programmer maintains, and it is external to the program. Is it possible to convert an integer number from upper case to lower case? Or to add two operations? The answer is yes, anything is possible in assembly, but that does not mean it makes sense to do it.

In assembly language it is important for the programmer to always be aware of what a series of bits in memory represents, or the context of the data. Remember that data without a context is never meaningful.

## Chapter 1. 9 Summary

In this chapter the concept of binary was introduced, as well as ways to represent binary data such as binary whole numbers, integers, and ASCII. Arithmetic and logical operations were defined for binary data. Finally, the chapter introduced the concept of a context, where the context defines the meaning of any binary data.

## Chapter 1. 10 Exercises

1) What are the following numbers in binary and hexadecimal?
    a. $13_{10}$
    b. $15_{10}$
    c. $25_{10}$
    d. $157_{10}$
    e. $325_{10}$
    f. $1096_{10}$

2) What are the following numbers in decimal?
    a. $10011100_2$
    b. $9C_{16}$
    c. $1F_{16}$
    d. $0C_{16}$
    e. $109A_{16}$

3) Give the value of the following numbers (IE. $2^{32}$ = 4G)
    a. $2^{16}$
    b. $2^{24}$
    c. $2^{29}$
    d. $2^{34}$
    e. $2^{31}$

4) Give the 2's complement form for each of the following numbers:
    a. 13
    b. -13
    c. 156
    d. -209

5) Do the following calculations using 2's complement arithmetic. Show whether there is an overflow condition or not. CHECK YOUR ANSWERS!
    a. 12 + 8 with 5 bits precision
    b. 12 + 8 with 6 bits precision
    c. 12 – 8 with 5 bits precision
    d. 12 – 8 with 6 bits precision

6) Do the following multiplication operations using bit shift operations. Check your answers.
    a. 5 * 4
    b. 13 * 12
    c. 7 * 10
    d. 15 * 5

7) What is the hex representation of the following numbers (note that they are strings):
    a. "52"
    b. "-127"

8) Perform the following multiplication and division operations using only bit shifts.
    a. 12 * 4
    b. 8 * 16
    c. 12 * 10
    d. 7 * 15
    e. 12 / 8
    f. 64 / 16

9) Explain the difference between a short-circuiting and non-short-circuiting logical expression. Why do you think these both exist?

10) In your own words, explain why the context of data found in a computer is important. What provides the context for data?

11) Convert the following ASCII characters from upper-case to lower-case. Do not refer to the ASCII table.
    a. 0x41 (character A)
    b. 0x47 (character G)
    c. 0x57 (character W)

12) Convert the following ASCII characters from lower-case to upper-case. Do not refer to the ASCII table.
    a. 0x 62 (character b)
    b. 0x69 (character i)
    c. 0x6e (character n)

13) Write the functions *toUpper* and *toLower* in the HLL of your choice (C/C++, Java, C#, etc.) The toUpper function converts the input parameter string so that all characters are uppercase. The toLower function converts the input parameter string so that all characters are lowercase. Note that the input parameter string to both functions can contain both upper and lower-case letters, so you should not attempt to do this using addition.

14) Implement a program in the HLL of your choice that converts an ASCII string of characters to an integer value. You must use the follow pseudocode algorithm:

```
 read numberString
 outputNumber = 0
 while (moreCharacters) {
   c = getNextCharacter
   num = c - '0';
   outputNumber = outputNumber * 10 + num;
 }
print outputNumber;
```