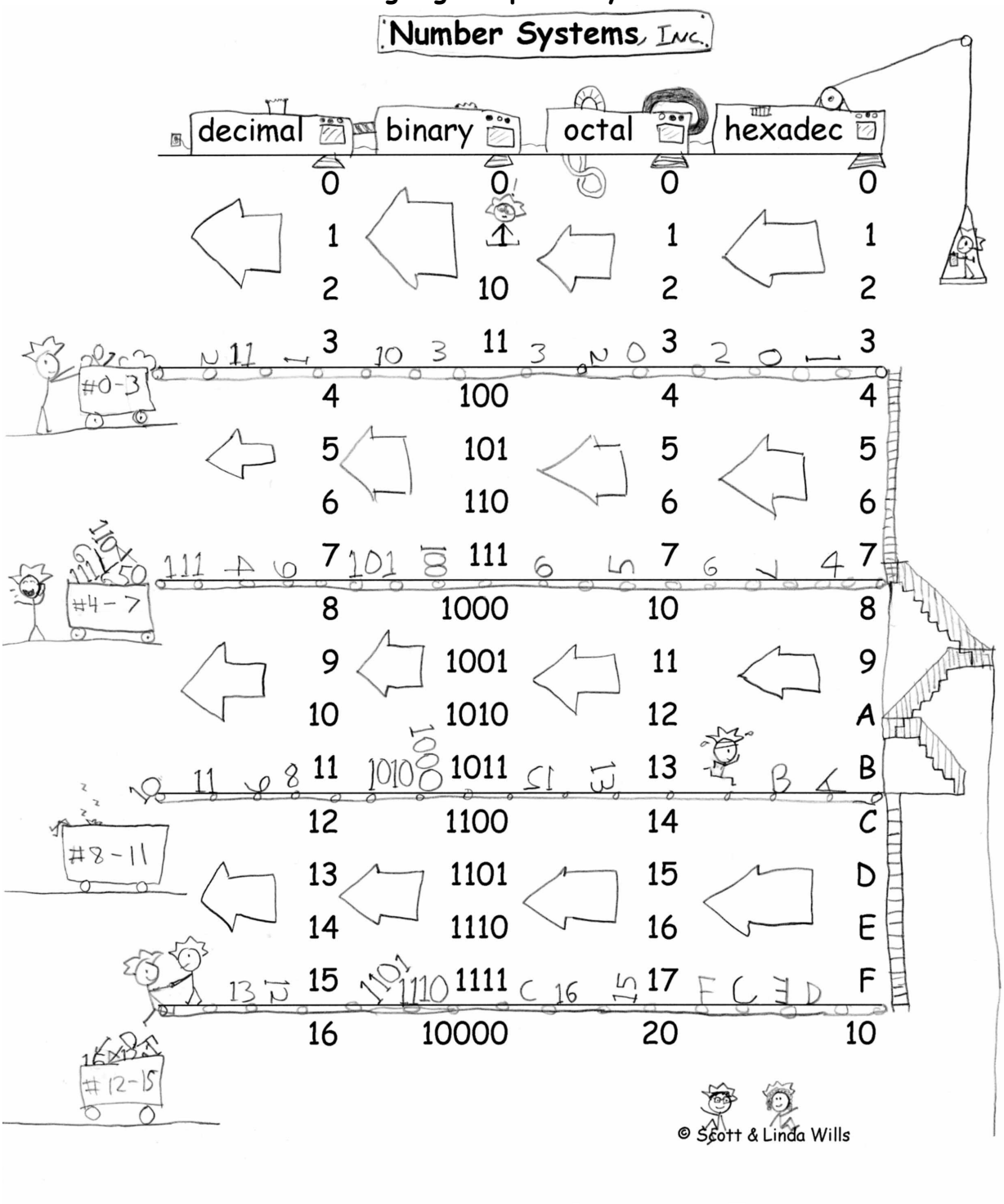
Designing Computer Systems

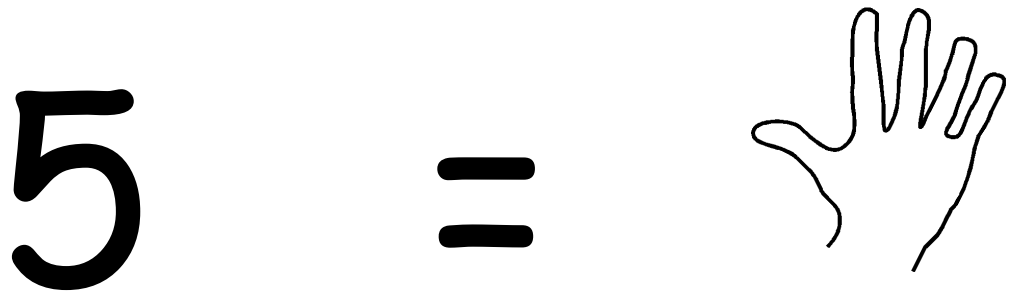


Designing Computer Systems

Number Systems

Most concepts are easier to learn when you're already familiar with them. But a few concepts are more difficult to learn because you know them so well. In our early childhood, we learn that abstract symbols represent real things in our world. The word "candy" represents something that tastes sweet. The word "bedtime" means you're about to leave the party. A symbol and its meaning are locked together in our brain.

This is especially true for qualitative symbols. Here we see the symbol "5" represents the quantity five. In fact, its difficult to describe the symbol with implying its meaning.



symbol meaning

But for computers, a symbol has no implicit meaning. It is a string of ones and zeros. Only when we instruct the computer on how to process a symbol does it have meaning. In many programming languages, you must declare the *type* of a variable, (i.e., an integer, a floating point, or a character string) before you can perform operations on it. This allows the compiler to assign the correct instruction for that interpretation of the variable's value.

Number systems separates a symbol and its meaning into two distinct concepts: a *notation* and a *representation*. Notations determine how symbols can be created using strings of characters from a given alphabet. Representations show how to assign real world meaning to a given string.

Native Notations: Humans around the world favor decimal (base 10) notation. An anthropologist might suggest this is because we have ten fingers. People define ten characters (0,1,2,3,4,5,6,7,8,9) to represent quantities. These characters form a notation *alphabet*. We use this alphabet to create multi-character *strings*, which provide a limitless number of intuitive, unique symbols. In base 10, a N character string can provide 10^N unique strings.

A computer also has a native notation. It uses binary (base 2) notation because the limited multiplicity of its "fingers" maintain digital states: 1 or 0, high or low, true or false. It also builds strings out of its two character alphabet (0 and 1). An N character binary string provides 2^N unique symbols.

Binary, requires longer strings to achieve the same number of symbols. A three character decimal string can represent 1000 symbols (000 - 999). It takes ten character binary string to achieve the same number of strings (0000000000 - 1111111111). To keep the length of written symbols manageable, we often use power of two bases octal (base 8) and hexadecimal (base 16).

The table below shows the ordered sequences in each notation. Notice that each digits counts through the base's alphabet. When a digit reaches the last character, it wraps back to zero and the next digit position is advanced. In all notations, leading zeros are implied, but not drawn.

decimal	binary	octal	hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Notational Conversion: Since all notations begin with zeros, strings on a row in the table are the same sequence number. Since we use the sequences in order, notational conversion of a string in one notation is accomplished by finding the

corresponding position in another notation. For example, the string 11 in decimal is 1011 in binary, 13 in octal, and B in hexadecimal. This conversion takes no position on the meaning of the string. Rather it shows string equivalence.

Since binary, octal, and hexadecimal are all power of two bases, they are more easily translated because they each can be represented as a whole number of binary digits or *bits*. Converting from one power of two notation to another is simply a matter of regrouping the bits. Here are a few examples:

10101110 (binary) = 010 101 110 = 256 (octal) = 1010 1110 = AE (hexadecimal)

153 (octal) = 001 101 011 = 1101011 (binary) = 0110 1011 = 6B (hexadecimal)

68A (hexadecimal) = 0110 1000 1010 = 0110 1000 1010 (binary) = 011 010 001 010 = 3212 (octal)

A conversion between a power of two bases (e.g., binary) and decimal is more complicated. A decimal digit is approximately three and a third bits, so bit regrouping will not work. Notational conversion between binary and decimal is accomplished by finding the string sequence position (how many strings is it from all zeros) and then converting the number between binary and decimal.

In an arbitrary base B, a N character string provides B^N unique symbols. The first digit on the right is the one's place. The second digit is the B's place, the third digit is the (B^2)'s place, the fourth digit is the (B^3)'s place etc. The familiar decimal places are 1s, 10s, 100s, 1000s, … In binary, the places 1s, 2s, 4s, 8s, 16s, … are less familiar, but more useful *powers of two*.

Powers of Two: When you work with computers, you must know the powers of two. Bad news: we have to memorize a few of them. Good news: we don't need to know very many. Here are the ones to learn:

$2^0 = 1$ $2^1 = 2$ $2^2 = 4$ $2^3 = 8$ $2^4 = 16$ $2^5 = 32$

$2^6 = 64$ $2^7 = 128$ $2^8 = 256$ $2^9 = 512$ $2^{10} = 1024 = \sim 1K$

Memorizing can be difficult … but not here. Most folks can compute through 2^4 in your head. 2^6 is 64. The sixes go together. 2^8 is 256. Eight bits is a byte so 256 shows up all the time. 2^5, 2^7, and 2^9 are either twice or half an easy one. And 2^{10} is the vehicle for all other powers of two! It is approximately 1000 (1K).

To find larger powers of two, recall that exponents can be reduced like this:

$$B^{X+Y} = B^X \cdot B^Y$$

We can break larger powers of two into groups in the table above. Exponent multiples of ten can be grouped to become 1000. Here are a few examples.

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

$2^{24} = 2^4 \times 2^{10} \times 2^{10} = 16 \times 1K \times 1K = 16M$

$2^{25} = 2^5 \times 2^{10} \times 2^{10} = 32 \times 1K \times 1K = 32M$

$2^{32} = 2^4 \times 2^{10} \times 2^{10} \times 2^{10} = 4 \times 1K \times 1K \times 1K = 4G$

$2^{41} = 2^1 \times 2^{10} \times 2^{10} \times 2^{10} \times 2^{10} = 2 \times (1K)^4 = 16T$

$2^{-18} = 2^{-8} \times 2^{-10} = 1 / (256 \times 1K) = 1 / 256K$

Binary to Decimal: Using powers of two, binary numbers can be converted using the place values. Here's an example:

64's	32's	16's	8's	4's	2's	1's
1	1	1	1	0	0	1

In a base, the order of a string in a notation is found by summing the products of each character and its respective digit's significance. In binary, the digit values are powers of two. Since characters are either 0 or 1, multiplication is easy. In this example, the corresponding decimal string is computed as:

$$1 \times 64 + 1 \times 32 + 1 \times 16 + 1 \times 8 + 0 \times 4 + 0 \times 2 + 1$$

$$64 + 32 + 16 + 8 + 1$$

$$80 + 40 + 1$$

121 (decimal)

Note that many of the powers of two sum to form multiples of ten. Here are a few more examples. The bases are indicated here with subscript.

$110110_2 = 32 + 16 + 4 + 2 = 54_{10}$ $\quad$ $10101010_2 = 128 + 32 + 8 + 2 = 170_{10}$

$100001000_2 = 256 + 8 = 264_{10}$ $\quad$ $1111_2 = 8 + 4 + 2 + 1 = 15_{10}$

A string of ones always sums to next place value minus one.

Decimal to Binary: Notational conversion from decimal to binary is similar. Only here you subtract away powers of two until you reach zero.

78_{10}		165_{10}		500_{10}	
- 64	1000000	- 128	10000000	- 256	100000000
14		37		244	
- 8	+ 1000	- 32	+ 100000	- 128	10000000
6		5		116	
- 4	+ 100	- 4	+ 100	- 64	1000000
2		1		52	
- 2	+ 10	- 1	+ 1	- 32	100000
0	1001110_2	0	10100101_2	20	
				- 16	10000
				4	
				- 4	100
				0	111110100_2

Often there are tricky ways to do things. Sometimes they help. Sometimes they don't. For decimal to binary conversion, one can simply perform a series of halvings (dividing by two). If the number being halved is an even number, list a "0". If the

number being halved is odd, subtract one and list a "1". When you reach zero, the list of ones and zeros is the binary notation. Let's try 78 and 165 this way.

78	~~39~~ 38	~~19~~ 18	~~9~~ 8	4	2	~~1~~ 0	
0	10	110	1110	01110	001110	1001110_2	
~~165~~ 164	82	~~41~~ 40	20	10	~~5~~ 4	2	~~1~~ 0
1	01	101	0101	00101	100101	0100101	10100101_2

This trick works by deconstructing the decimal value from its binary components, from least significant to most significant. It gives the right result; but it sometimes requires more calculations and it is harder to double check the result.

It may appear that integer values are being translated between different bases. But we are only finding corresponding strings in different bases. Notations do not imply meaning.

Get to the Point: Sometimes strings include a point (a decimal point in base 10) as part of the notation. This point divides the string into two parts, a substring to the left of the point and a substring to the right. When performing notation conversion, start at the point and work left and then right. This addresses unwritten leading and trailing zeros. Let's try a few power of two conversion examples.

10100101.011011_2 = 1010 0101 . 0110 1100 = $A5.6C_{16}$

1101001.1111_2 = 001 101 001 . 111 100 = 151.74_8

$26.BC_{16}$ = 0010 0110 . 1011 1100 = 101 110 . 101 111 = 56.57_8

46.26_8 = 100 110 . 010 110 = 0010 0110 . 0101 1000 = 26.58_{16}

Sometimes leading and trailing zero are adding and subtracted to form necessary bit groupings. But notice that they always work out, left and right, from the point. Binary to decimal conversions with a point is the same, only the bit positions are fractions.

4's	2's	1's		1/2's	1/4's
1	0	1	.	1	1

4 + 1 + .5 + .25 = 5.75

8's	4's	2's	1's		1/2's	1/4's	1/8's	1/16's
1	0	1	0	.	0	1	0	1

8 + 2 + .25 + .0625 = 10.3125

8's	4's	2's	1's		1/2's	1/4's	1/8's	1/16's
1	1	0	1	.	1	0	1	1

8 + 4 + 1 + .5 + .125 + .0625 = 13.6875

Representations - Finding Meaning in a Digital World: Although the use of a point has implications to a sequence's value, the focus thus far has been on notational conversion. A given sequence is composed of a specified numbers of characters (N) in a given base (B) offering B^N unique codes. How those codes are used is dependent on *representations*.

Unsigned Integers: A representation begins with a requirement: what needs to be represented. Suppose a digital system is counting objects being manufactured in a factory. The counting numbers (0, 1, 2, …) are needed to maintain a tally. These unsigned integers can be associated with notational sequences in an intuitive way.

sequence	meaning
0000	"0"
0001	"1"
0010	"2"
0011	"3"
0100	"4"
0101	"5"
0110	"6"
0111	"7"

sequence	meaning
1000	"8"
1001	"9"
1010	"10"
1011	"11"
1100	"12"
1101	"13"
1110	"14"
1111	"15"

Perhaps this is too intuitive, since this looks like notational conversion from binary to decimal. But here the quoted value really does mean a quantity (remember the fingers). A four bit binary sequence is used to represent a quantity between "0" and "15". In general, when representing unsigned integers, an N-bit binary sequence can represent quantities between "0" and "2^N -1". So an eight bit unsigned integer can represent quantities between "0" and "255"; a 16 bit unsigned integer can represent "0" to "65,535" (around 64K), and a 32 bit unsigned integer can represent "0" to "4 billion". This process is nothing more than a uniform value sequence assignment. An integer value is assigned to each sequence.

Signed Integers: Some applications require negative as well as positive integers. While it doesn't have to be this way, a signed representation typically offers an equal number of positive and negative quantities.

signed	sequence	unsigned
"0"	0000	"0"
"1"	0001	"1"
"2"	0010	"2"
"3"	0011	"3"
"4"	0100	"4"
"5"	0101	"5"
"6"	0110	"6"
"7"	0111	"7"

signed	sequence	unsigned
"-8"	1000	"8"
"-7"	1001	"9"
"-6"	1010	"10"
"-5"	1011	"11"
"-4"	1100	"12"
"-3"	1101	"13"
"-2"	1110	"14"
"-1"	1111	"15"

Since half of the sequences are used to represent negative values, there are not as many to represent positive quantities. Here the 16 sequences represent "-8" to "+7". In general, this N-bit signed integer representation can represent quantities from "$-2^{(N-1)}$" to "$2^{(N-1)} - 1$". A eight bit signed integer can represent "-128" to "+127". A 16-bit signed integer can represent "-32,678" to "+32,767" (±32K). A 32-bit signed integer can represent "±2 billion" (±32G). Why isn't it symmetric? Because zero has to go somewhere (and use a sequence). Here it is counted as a positive value. This signed representation is called *two's complement*.

There are many choices for signed representations. But only one, two's complement is widely used, and for good reasons. As number systems and arithmetic are explored, two's complement has many significant advantages other other signed representations.

- Sign and Magnitude: This signed representation (used in floating point) employs all but one bits for an unsigned magnitude. The remaining bit indicates the sign. It problems include complex arithmetic logic (since addition sometimes becomes subtraction and vice versus) and two representations of zero (+0 and -0). This may seem like a small matter. But comparison to zero is the most commonly performed conditional operation. If there are two values representing zero, this operation become more complex.
- One's Complement: This signed representation has a simple negation: complement each bit. So +1 (0001) is negated to -1 (1110). This representation also introduces complexity is arithmetic. And it has two values for 0 (0000) and (1111).

Two's complement is related to one's complement. Negation involves complementing each bit in the representation. But then one is added: one's complement + one = two's complement. It only has one representation of zero (negating zero give zero). Sign is easy to determine; the most significant bit of the representation indicates the sign (0 = positive, 1 = negative). But it is not a sign bit. And arithmetic using two's complement couldn't be easier (one can ignore sign). Two's complement also works well with non-integer representations, which come next.

Fixed Point: Integer representations have a *fixed step size*, the value one. All adjacent sequences differ by the integer value one. This is its *resolution* and it is fixed. This step size can assume any value, depending on the position of the point (which separates whole and fractional parts of the representation). So if the point is fixed one bit position to the left of integers, the step becomes 0.5 instead of one. This four-bit, fixed point representation offers a different set of values.

signed	sequence	unsigned
"0.0"	000.0	"0.0"
"0.5"	000.1	"0.5"
"1.0"	001.0	"1.0"
"1.5"	001.1	"1.5"
"2.0"	010.0	"2.0"
"2.5"	010.1	"2.5"
"3.0"	011.0	"3.0"
"3.5"	011.1	"3.5"

signed	sequence	unsigned
"-4.0"	100.0	"4.0"
"-3.5"	100.1	"4.5"
"-3.0"	101.0	"5.0"
"-2.5"	101.1	"5.5"
"-2.0"	110.0	"6.0"
"-1.5"	110.1	"6.5"
"-1.0"	111.0	"7.0"
"-0.5"	111.1	"7.5"

For both unsigned and signed representations, there are the same number of sequences. With a smaller resolution (0.5 versus 1), the representation has a smaller range. In general, an N bit fixed point representation with K bits to the right of the binary point has a step size of $1/2^K$ and a range of $-2^{(N-1)}/2^K$ to $(2^{(N-1)}-1)/2^K$. The range is *divided* by the step size.

If the fixed point is set two bits from the left, the step size and range change. A smaller step, 0.25, yields higher resolution, but a smaller range.

signed	sequence	unsigned
"0.0"	00.00	"0.0"
"0.25"	00.01	"0.25"
"0.5"	00.10	"0.5"
"0.75"	00.11	"0.75"
"1.0"	01.00	"1.0"
"1.25"	01.01	"1.25"
"1.5"	01.10	"1.5"
"1.75"	01.11	"1.75"

signed	sequence	unsigned
"-2.0"	10.00	"2.0"
"-1.75"	10.01	"2.25"
"-1.5"	10.10	"2.5"
"-1.25"	10.11	"2.75"
"-1.0"	11.00	"3.0"
"-0.75"	11.01	"3.25"
"-0.5"	11.10	"3.5"
"-0.25"	11.11	"3.75"

Fixed point does not require a change to the arithmetic. It is only a matter of interpretation of the operands and the result. Fixed point is the presentation of choice for the financial world. All calculations must be accurate to the penny, regardless of the amount. This fixed resolution limits the range. Science and engineering often need something else.

Floating Point: Fixed point presentations have a problem in that their accuracy (the number of significant figures) is dependent on the magnitude of the represented value. The integer value 23,415,823 may have eight significant figures. But 16 has only two. Floating point has a different, more complex approach. Use a certain number of bits to represent the magnitude (the significant figures) of a value. Then use addition bits to scale it to the correct value. Most people have used this approach in scientific notation. The magnitude 6.022 is scaled by 10^{23} to express the number of molecules in a mole. This value would be difficult to express using a fixed point representation.

Floating point breaks the bits of the representation into fields: sign, mantissa, and exponent.

sign	mantissa	exponent

The sign field is a one bit field indicating the sign of the mantissa. This sign and magnitude representation makes sense when scaling the value. The mantissa is the largest field and contains the bits that provide the accuracy (significant figures) to the value being represented. Since the mantissa does not need to provide the scaling, its range is between zero and one. The exponent field is a signed integer that scales the mantissa to the proper value. In binary, the exponent is raised to a power of two, not ten. In general, a floating point value is computed as:

$$\text{sign} \times \text{mantissa} \times 2^{\text{exponent}}$$

where the sign is ±1, the mantissa is an unsigned fixed point value with the binary point at the right end of the sequence (K = N), and the exponent is a signed integer. Typical field lengths for an IEEE single precision floating point value is sign = one bit, mantissa = 23 bits, and exponent = 8 bits. This means that the unscaled step size is 1/8M of the mantissa. To find the equivalent decimal significant figures, consider the mantissa range (0 to 8,000,000). The first six digits can assume any value (0-9). The seventh decimal digit can assume 0-8. So this mantissa maintains between six and seven decimal significant figures. In general, every ten bits of mantissa provides three decimal significant figures.

The exponent field is a signed (two's complement) integer. Like scientific notation, it scales the mantissa to the proper value. It doesn't change the bits, rather it moves the binary point. Moving it right by one bit multiples the value by two. Moving right two bits multiples by four. Moving right by I bits multiples by 2^I. Moving left is similar, except it divides by 2^I. Because of this exponential scaling, a modest range in the exponent field can have an enormous effect on the value. An eight bit exponent has a range of -128 to +127. Since the mantissa is between zero and one, the final value an be as large as 2^{127} or as minuscule as $1/2^{128}$.

Floating points representations can assume smaller and larger number of bits. IEEE double precision floating point employs 64 bits including an eleven bit exponent and a 52 bit mantissa for approximately 15 significant figures. A 16 bit floating points might have a 10 bit mantissa (three significant figures) and a five bit exponent for values from 2^{15} (32K) to $1/2^{16}$ (1/64K).

Arithmetic operations in floating are more complicated since exponents must be adjusted before simple addition and subtraction can be performed in the mantissa. Afterwards, a process called *normalization* must be performed where the mantissa

and exponent are adjusted to keep a one in the most significant bit of the mantissa. This is necessary to maintain the full accuracy of the value.

In floating point, all values have a fixed accuracy (significant figures), but a varying resolution (step size). This contrasts with fixed point that has a fixed resolution, and a varying accuracy. Fixed point works for financial calculations. Floating point works for science and engineering. Both are important.

Full Disclosure: Floating point standards have many subtle complexities that are not covered here. For example, since normalization maintains a one on the most significant bit of the mantissa, it can be assume to effectively *add* a bit. Other field combinations are used for rare but important values like NaN (Not a Number). If interested, check out http://grouper.ieee.org/groups/754/.

Symbolic Values: Speaking of not a number, there is a large class of representation that don't represent quantities. Take this document, for example. Each character represents a letter of the alphabet, and sequences are strings of letters forming words, sentences, and paragraphs. One of the oldest and most common symbolic representation is ASCII (American Standard Code for Information Interchange). This seven bit representation includes the characters that appear on a keyboard: A-Z, 0-9, a-z, characters for punctuation, special symbols, etc. Plus some obsolete control characters like bell, ACK/NAK, etc. that date back to an era when mechanical teletypes were used to display text. This standard was latter expanded to eight bits (256 symbols) for CP/M, MS-DOS, etc. but it still lives on.

One limitation of ASCII is its inability to expand to international character sets. A modern alternative is Unicode, a 16-bit character code that embraces the diversity of symbols from around the world. While its larger 16 bits versus eight bits, its ability to international character sets justifies the extra storage. Still, ASCII is far from gone. It still is the primary representation used in text files under today's operating systems including Microsoft Windows, Mac OS X, and Linux.

Other Representations: There are hundreds of other representations to represent images (e.g., JPEG), videos (e.g., XviD), audio (e.g., mp3), vector graphics (e.g., postscript), and many other things. However the notations used generate the same patterns of sequences.

Summary: In digital computers, information is expressed in one of several notations, and its meaning is defined by one of many representations.

- Today's notations include binary, decimal, and hexadecimal. Powers of two fit the binary technology being used. Decimal fits ten fingered humans.

- Quantitative representations include signed and unsigned fixed point representations integers is a special case). For signed representations, two's complement is the representation of choice. Fixed point has a fixed step size (resolution), but varying accuracy. Floating point is a more complex representation with fixed accuracy, but a varying step size. Both representations have their place in digital systems.
- Symbolic representations are widely used in digital systems. ASCII is an old but widely used standard. Unicode allow representation of international characters.

ASCII Codes								
	0x00	0x10	0x20	0x30	0x40	0x50	0x60	0x70
0x0	NUL	DLE	SP	0	@	P	`	p
0x1	SOH	DC1	!	1	A	Q	a	q
0x2	STX	DC2	"	2	B	R	b	r
0x3	ETX	DC3	#	3	C	S	c	s
0x4	EOT	DC4	$	4	D	T	d	t
0x5	ENQ	NAK	%	5	E	U	e	u
0x6	ACK	SYN	&	6	F	V	f	v
0x7	BEL	ETB	'	7	G	W	g	w
0x8	BS	CAN	(	8	H	X	h	x
0x9	HT	EM	)	9	I	Y	i	y
0xA	LF	SUB	*	:	J	Z	j	z
0xB	VT	ESC	+	'	K	[	k	{
0xC	FF	FS	,	<	L	\	l	|
0xD	CR	GS	-	=	M	]	m	}
0xE	SO	RS	.	>	N	^	n	~
15	SI	US	/	?	O	_	o	DEL

