Data Representation

0.1 Introduction

This note is devoted to the different types of data supported by current computers. The following important types are described, character, boolean, signed integer, floating-point, fixed-point, and BCD (binary coded decimal). Also discussed are the concepts of carry and overflow. The non-numeric data types dealt with are character and boolean. First, however, let's try to understand the meaning of the term "data type." A data type is something like "integer", "real", or "boolean." These words are used all the time in higher level programming and most users have a vague, intuitive understanding of them. It is possible, however, to assign them a precise meaning. Recall, for example, that a boolean quantity can take one of two values, namely "true" or "false." We say that the set of all boolean values is of size 2, i.e. $\{0, 1\}$. This leads to the definition: A *data-type* is a set of values.

0.2 Boolean

A Boolean is a word derived from the name of George Boole, an Irish mathematician and philosopher who founded the field of modern symbolic logic. He was interested in operations on numbers that result in non-numeric quantities, such as $a > b$, that results in either true or false, and came up with a formalism of such operations, now called as boolean algebra. His work is widely used in the design of computers, since all the arithmetic operations in the computer are performed in terms of logical operations. The boolean values in computer are represented as 1 and 0, or 1xxxxxx and 0xxxxxx, where x's are don't care bits, meaning that, they can be either 0 or 1 or combinations.

0.3 Numeric Data Types

Following are the factors for selection of type number representation in computers,

1. types of numbers to be represented: integers, real numbers, complex numbers
2. range of values to be dealt with
3. precision of numbers required
4. cost of hardware required to store and process these numbers.

The numbers are represented in one of the two *formats: fixed point* and *floating point* format. First has small range and simple hardware requirements while second requires complex hardware, but has higher range of representable numbers.

Most computers include two data types in this category, namely integer and real. However, this, section shows that it is possible to define other numeric data types that can be useful in special situations.

0.3.1 Signed Integers

The binary number system was discovered by the great mathematician and philosopher Gottfried Wilhelm Von Leibniz on March 15, 1679. This representation of the integers is familiar and is the most "natural" number representation on computers in the sense that it simply uses the binary values of the integers being represented. The only feature that needs to be discussed, concerning integers, is the way signed integers are represented internally. *Unsigned* numbers are certainly not enough for practical calculations, and any number representation should allow for both positive and negative numbers. Three methods have been used throughout the history of computing to represent signed integers. These are *sign-magnitude*, *one's complement*, and *two's complement*. All three methods reserve the leftmost bit for the sign of the number, and all three use the same sign convention, namely 1 represents a negative sign and 0 represents a positive sign.

0.3.2 Sign-Magnitude Representation

In this method the sign of a number is changed by simply complementing the sign bit. The magnitude bits are not changed. To illustrate this method we use 4-bit words, where one bit is reserved for the sign, leaving

three magnitude bits. Thus 0|111 is the representation of +7 and 1|111, that of −7. The number 0|010 is +2 and 1|010 is −2. In general, 1xxx is the negative value of 0xxx. The largest number in our example is +7 and the smallest one is −7. In general, given n-bit words, where the leftmost bit is reserved for the sign, leaving $n-1$ magnitude bits, the largest possible number is $2^{n-1}-1$ and the smallest one is $-(2^{n-1}-1)$. The zero is both positive and negative here.

This representation has the advantage that the negative numbers are easy to read but, since we rarely have to read binary numbers, this is not really an advantage. The disadvantage of this representation is that the rules for the arithmetic operations are not easy to implement in hardware. Before the ALU (arithmetic and logic unit) can add or subtract such numbers, it has to compare them, in order to decide what the sign of the result should be? When the result is obtained, the ALU has to append the correct sign to it explicitly. The sign-magnitude method was used on some old, first-generation computers, but is no longer being used.

For fractional numbers, the sign magnitude scheme reserve a bit for sign and a bit string following to that for magnitude in integer and fractions. For example,

$$-1101.0101_2 = -13.3125_{10}$$

This is because, fraction $0.0101_2 = \frac{1}{2} \times 0 + \frac{1}{4} \times 1 + \frac{1}{8} \times 0 + \frac{1}{16} \times 1 = 0.3125_{10}$, accordingly, there is a decimal equivalent.

For non-negative (unsigned)integer representation there is no need to bother for sign. Hence, 8-bit binary number can represent $0_{10} \ldots 255_{10}$. If we consider that the binary number is $A = a_{n-1}a_{n-2}\ldots a_0$, then, its equal decimal number is

$$A = \sum_{i=0}^{n-1} 2^i a_i \tag{1}$$

For Signed magnitude representation using binary format, since the left most bit (most significant bit) is reserved for sign, hence, for 8-bit size memory, only 7-bits shall be available for magnitude part. Therefore, $+18_{10} = 00010010_2$ in 8-bit representation, and $-18_{10} = 10010010_2$. As an expression, we write,

$$A = \sum_{i=0}^{n-2} 2^i a_i, \text{ if } a_{n-1} = 0 \tag{2}$$

and

$$A = -\sum_{i=0}^{n-2} 2^i a_i, \text{ if } a_{n-1} = 1. \quad (3)$$

Following are the drawback of sign-magnitude representation:

1. it requires consideration of sign and magnitude separately,
2. there is double notation for 0, i.e., +0 and −0, i.e., $+0_{10} = 00000000$, and $-0_{10} = 10000000$.
3. for doing arithmetics, you need to compare them, and give a sign in result as per the magnitude of the larger number.

0.3.3 One's Complement

This method is based on the simple concept of *complement*, i.e. inversion of bits of a binary number. It is more suitable than the sign-magnitude method for hardware implementation. The idea is to represent a negative number by complementing the bits of the original, positive number. This way, we hope to eliminate the need for a separate subtraction circuit in the ALU and subtract the numbers by adding their complement. Perhaps the best way to understand this method is to consider the complement of a decimal number, as given in the following example.

Example 0.3.1 *Subtraction using 9's complement.*

Instead of performing the subtraction on decimal numbers: $12845 - 3806$, we add complement of decimal 3806 into 12845. The first step is to complement the second number. The most natural way to complement a decimal number is to complement each digit with respect to 9 (the largest decimal digit). Thus the complement of 03806 would be 96193. The second step is to add the two numbers, which yields the sum 12845 + 96193 = 109038. This is truncated to six digits (one sign digit plus five magnitude digits), to produce 09038. The correct result, however, is +9039, which is obtained by artificially adding a 1 to our result. But before accepting this fact, the reader may try some more examples to convince herself that this works often, but not always, when numbers with different signs are added. If the net result is negative, there would not be any carry like this. □

Example 0.3.2 *Subtraction and addition using 1's complement.*

Consider subtraction of 25 from 17. Their binaries are, 11001 and 10001, respectively. After adding the 1's complement of second number, we get 11001 + 01110 = 100111. Removing the left most bit in the result and adding into the rest, we get result $00111 + 1 = 01000_2 = 8_{10}$. □

The one's complement method was used on many second generation computers, but was rejected in favor of the newer two's complement method.

0.4 Two's Complement representation

Using 2's complement representation, negative numbers are represented as two's complement, and positive numbers have zero in the most significant bit (MSB) position. In this notation, the most significant bit is used as sign bit, while other bits are treated differently. For n-bit binary number, the range of values which can be represented is -2^{n-1} to $+2^{n-1} - 1$. Also, the number of representations of zero is one only. The addition and subtractions are straight forward. If the result of calculation is negative (the MSB is 1) then actual result's magnitude is found by obtaining its two's complement. Apart from this there is important symmetry - the boundary numbers are complement of each other, i.e., -2^{n-1} and $+2^{n-1} - 1$. The equation 4 gives an expression for 2's complement representation for both positive and negative numbers. For $a_{n-1} = 0$, the term $-2^{n-1}a_{n-1}$ is zero and the equation defines a non-negative number. For $a_{n-1} = 1$, the term 2^{n-1} is subtracted from the summation, resulting to a negative number.

$$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \tag{4}$$

The table 1 shows the values for two representation schemes, i.e., sign magnitude and two's complement.

Table 1: Sign-magnitude v/s 2's complement mode.

Decimal Representation	Sign Magnitude	Two's Complement
+7	0111	0111
+6	0110	0110
+5	0101	0101
+4	0100	0100
+3	0011	0011
+2	0010	0010
+1	0001	0001
+0	0000	0000
-0	1000	0000
-1	1001	1111
-2	1010	1110
-3	1011	1101
-4	1100	1100
-5	1101	1011
-6	1110	1010
-7	1111	1001
-8	None	1000

0.5 Floating point Representation

Since integers are not sufficient for all calculations, computer designers have developed other representations where non-integers can be represented and operated on. The most common of which is the floating point representation (fp for short), normally called real. In addition to representing non-integers, the floating point method can also represent very large and very small (fractions very close to zero) numbers. The method is based on the common scientific notation of numbers (for example, 56×10^9) and represents the real number x in terms of two signed integers a (the mantissa) and b (the exponent) such that $x = a \times 2^b$ (figure 1).

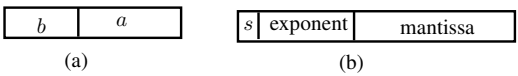

Figure 1: Floating-point representation, (a) as two binary numbers, (b) with a sign-bit.

Table 2: Binary numbers in fixed-point format(fig. 1(a))

	a	b	$a \times 2^b$	value
1.	1	1	1×2^1	2
2.	1	10	1×2^{10}	1024
3.	1111	-3	15×2^{-3}	1.875
4.	1	-10	1×2^{-10}	≈ 0.001
5.	100	-1	100×2^1	50

The table 2 shows some simple examples of real numbers that consist of two integers. Note that none of the integers is very large or very small, yet some of the floating point numbers obtained are extremely large or are very small fractions. Also some floating point numbers are integers, although in general they are not.

The table 2 shows two important properties of floating point numbers. One is that magnitude of the number is sensitive to the size of the exponent b; the other is that the sign of b is an indication of whether the number is large or small. Values of b in the range of 20 result in floating point numbers in the range of a million, and a small change in b (as, for example, from 20 to 21) doubles the value of the floating point number. Also, floating point numbers with positive b tend to be large, while those with negative b tend to be small.

The function of the mantissa a is not immediately clear from the table, but is not hard to observe. The mantissa contributes the significant digits to the floating point number.

More insight into the nature of the mantissa is gained when we consider how floating point numbers are multiplied. Given the two floating point numbers $x = a \times 2^b$ and $y = c \times 2^d$, their product is $x \times y = (a \times c) \times 2^{b+d}$. Thus, to multiply two floating point numbers, their exponents have to be added, and their mantissas should be multiplied. This is easy since the mantissas and exponents are integers, but it involves two problems:

1. The sum $b + d$ may overflow. This happens when both exponents are very large. The floating point product is, in such a case, too big to fit in one word (or one register), and the multiplication results in overflow. In such a case, the ALU should set the V flag (for overflow), and the program should test the flag after the multiplication, before it uses the result.

2. The product $a \times c$ is too big. This may happen often because the

product of two n-bit integers can be up to $2n$-bits long. When this happens, the least-significant bits of the product $a \times c$ should be cut off, resulting in an approximate floating point product $x \times y$. Such truncation, however, is not easy to perform when the mantissas are integers, as the next paragraph illustrates.

Suppose that the computer has 32-bit words, and that a floating-point number consists of an 8-bit signed exponent and a 24-bit mantissa (we ignore the sign of the mantissa). Multiplying two 24-bit integer mantissas produces a result that's up to 48-bits long. The result is stored in a 48-bit temporary register, and its most-significant 24-bits are extracted, to become the mantissa of the product $x \times y$.

Computer designers solve this problem by considering the mantissa of a floating-point number a fraction. Thus, a mantissa of $10110\ldots0$ equals to $0.1011_2 = 2^{-1} + 2^{-3} + 2^{-4} = 11/16$. The mantissa is stored in the floating point number as $10110\ldots0$, and the (binary) point is assumed to be to the left of the mantissa. Normalization of floating-point numbers is introduced below.

0.5.1 IEEE754 Floating-point Format

A decimal number 9760000 can be represented in floating point as 0.97×10^7, and a fractions 0.0000976 can be represented as 0.976×10^{-4}, in general,

$$\pm S \times B^{\pm E} \tag{5}$$

here **S** is called *significand* (i.e., mantissa), **B** is *base*, which is 10 for decimal, and 2 for binary numbers. The figure 2 shows the structure of general format for IEEE754. The representation means, $(-1)^{sign} \times (1 + mantissa) \times 2^{expo.-bias}$

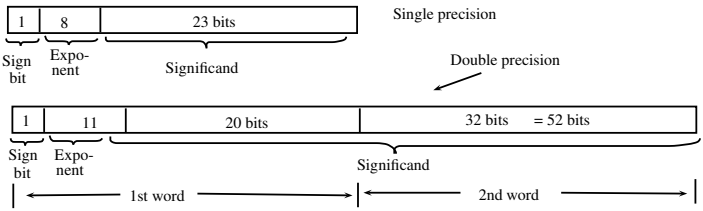

Figure 2: IEEE754 Floating-point Format.

0.5.2 Biased Representation

The two's complement is good for addition and subtraction operations on positive and negative numbers with or without fractions. However, it is not efficient for exponential format. This is because, the two's complement exponent is not effective for sorting. For example, -1 = 1...11, +1=0...01), where a negative number has higher absolute value. Hence, these number cannot be sorted.

To resolve the above problem, a bias is introduced so that 0...0 is smallest representable exponent, and 1 ...1 is largest. For a k-bit exponent, a bias value of $2^{k-1}-1$ is subtracted from the exponent to obtain the true value of exponent. For *single precision*(SP), the bias is 127, and for double precision it is 1023. The range for SP is -2×10^{-38} to 2×10^{38}, while for DP it is -2×10^{-308} to 2×10^{308}.

To handle the fractions, for example, in scientific notations, there is no leading 0, hence an implicit leading 1 is taken in mantissa. The *Significand* is taken as 24-bits, and the most significant bit is always taken as 1. Thus, 32-bit biased format is,

$$\pm 1.bbbb\cdots\times 2^{\pm E} \tag{6}$$

Here b's are binary, which are 0s and 1s. Some examples are given in the following. Consider a 32-bit number in biased exponent format as,

$$\{1\}\{01101011\}\{10100010000000000000000\}$$

This number is -1.1010001×2^x, where $x = 01101011 = 107_{10}$, true value of exponent is $x + 2's$ complement of 127 (i.e., $01101011_2 + 10000001_2$) $= -00010100 = -20$. Therefore, true value of above FP representation is -1.6328125×2^{-20}. The 23-bits mantissa provides precision equivalent to 8 decimal digits, and 53 bit mantissa to 16 decimal digits.

0.5.3 Extended single precision and double precision

Following are some additional features of IEEE754 format, which are an extension of standard floating point representation.

When exponent is 0 and mantissa is also 0, with sign bit, it is treated as ±0.

When exponent consists of all 1's and mantissa is 0, it represents ∞. An infinity is caused due to over-flow, i.e., due to division by 0. With sign bit it is$\pm\infty$.

When, exponent is 0, and nonzero mantissa, it is called *denormal* number. The value for this is $\pm0.M\times2^{-126}$. These numbers are smaller than smallest normal (representable) number.

When $E = 255$, $M \neq 0$, it is *Nan* (Not a number format) indicator, e.g. due to operation of $0/0$ or $\sqrt{-1}$.

0.5.4 Floating Point Addition/subtraction algorithms

Algorithm-to-add two FP numbers:

1. normalize the numbers: align the two numbers with respect to decimal point, inserting zeros after the decimal in smaller number and at the same time increase its exponent, until the exponents in two becomes equal
2. Add/subtract significands (mantissas)
3. Normalize the sum
4. Check for overflow or underflow and raise an exception if necessary

Arithmetic:

- Let $X = X_s \times B^{X_E}$, and $Y = Y_s \times B^{Y_E}$
- Then, $X + Y = (X_S \times B^{X_E - Y_E} + Y_S) \times B^{Y_E}$, if $Y_E \geq X_E$

  $X - Y = (X_S - Y_S \times B^{Y_E - X_E}) \times B^{X_E}$, if $X_E \geq Y_E$
- Let $X = 0.3 \times 10^2 = 30$ and $Y = 0.2 \times 10^3 = 200$. Then, $X + Y = (0.3 \times 10^{2-3} + 0.2) \times 10^3 = (0.03 + 0.2) \times 10^3 = (0.23) \times 10^3 = 230$.

0.6 Floating Point Multiplication and Division algorithms

Algorithm-to-multiply

1. Add the exponents and subtract bias
2. Multiply significands
3. Normalize if necessary
4. Check for over-/underflow and raise exception if needed
5. Round significand
6. Set the sign to positive if input signs are equal, negative if they differ

The divide arithmetic is similar to multiplication; the division takes place for two mantissa and subtraction is done for exponents.

Arithmetic:

- Let $X = X_s \times B^{X_E}$, and $Y = Y_s \times B^{Y_E}$

  $X \times Y = (X_S \times Y_S) \times B^{X_E + Y_E}$

  $X \div Y = \frac{X_S}{Y_S} B^{X_E - Y_E}$

0.6.1 Computer arithmetic v/s arithmetic in mathematics

Computers, of course, are commonly used for numerical calculations. However, computer arithmetic is different from the arithmetic that we study in school, in two ways: (1) computers have finite capacity and (2) they operate at a finite (albeit high) speed. The finite capacity means that certain numbers (very large numbers, very small ones, and numbers with many significant digits) cannot be stored in the computer. If such a number is generated during calculations, only an approximate value can be stored. If the number is a final result, this may not be a serious problem. If, however, it is an intermediate result, needed for later calculations, its approximate value may affect the final result seriously and we may end up with a wrong result. This is especially noticeable in floating point calculations, where the hardware may generate approximate results, store them in memory, and use them in later calculations, without any error message or even a warning.

It is well known that any *numerical algorithm* must be *stable*. If a numerical method is not stable, small arithmetic errors, resulting from the use of approximate numbers, may accumulate and cause a wrong final result.

The other difference, the finite speed of the computer, is both a blessing and a curse. Computers are, of course, much faster than humans and make possible many calculations that, in the past, were impossible because of the amount of work involved. On the other hand, there are calculations that, even on the most powerful computers, may take many years to accomplish. Two familiar examples are the search for large prime numbers and code breaking.

Even though prime numbers do not have many practical uses, mathematicians keep searching for larger and larger ones. Since there are infinitely many prime numbers, there is a limit even to the power of the fastest computers to find large primes. Secret codes are certainly very

practical. Cryptographers are engaged in developing secure codes but, at the same time, crypt-analysts are kept busy trying to break those very codes. It is a common belief today that there is no absolutely secure code that will also be practical (although impractical codes may be fully secured). Any code can be broken if it is used enough times, given enough expertise and enough computer time. In practice, however, a code that takes a year of computer time to break can be considered very secure. Such codes exist and are an example of the limitations put on computer arithmetic because of the limited speed of computers.

0.7 ASCII Code

The 7 bits/character is a good size for a character code. This, however, ignores one important aspect of character codes, namely reliability. Character codes are stored in memory, moved inside the computer, sent along computer buses to I/O devices, and even transmitted over long distances between computers. As a result, errors can creep in, so a well-designed character set should pay attention to code reliability. A very simple way of increasing the reliability of a code is to add one more bit, called a parity bit, to each code. The parity bit is chosen such that the total number of 1's in the code is always even, called *even parity*. For *odd parity* system the total number of 1's always odd.

Thus the (odd) parity of the group 10110 is 0, since the original group plus the parity bit has an odd number (3) of 1's. It is also possible to use even parity. For even parity, the code 1101001, that has four 1's to begin with, is assigned a parity bit of 0, while the code 1101011, with five 1's, gets a parity bit of 1, to complete the number of ones to an even number.

If a single bit gets corrupted, because of electrical or some other interference, a check of the parity bit will reveal it.

Hence, when a character is received by the computer as input, parity is checked. Often, when a character is loaded from memory, the memory hardware also checks parity. Obviously, a single parity bit cannot detect every possible error. The case where two bits get bad is the simplest example of an error that cannot be detected by parity. The result of adding a parity bit is that the ideal code size is now 8 (or rather 7+1) bits/character. This is one reason why most computers have a word size that's a multiple of 8.

The term ASCII (American Standard Code for Information Interchange) follows the rule of 7 bits for code, and one bit is left for adding parity. This code was developed in the early 1960s by the American National Standards Institute (ANSI Standard No. X3.4) and is based on

older telegraph codes. The figure 3 shows the ASCII table.

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

Figure 3: ASCII-table

The ASCII encodes 128 specified characters-numbers $0-9$, letters $a-z, A-Z$, some punctuation symbols, some control codes originated with *Teletype machines*, a blank space, into 7-bit binary integers. ASCII represent text in computers, communications equipment, and other devices that use text.

For the ASCII Codes, the first 32 codes are control characters. These are commands used in input/output and communications, and have no corresponding graphics, i.e., they cannot be printed out. In addition, the codes 20_{16} and $7F_{16}$ are also control characters.

The particular codes are arbitrary. The code of A is 41_16, but there was no special reason for assigning this. The B has the code 42_{16}, C has 43_16, etc.

There is also a simple relationship between the codes of the uppercase and lowercase letters. The code of a is obtained from the code of A by setting the most significant (7th) bit to 1.

The parity bit (8th) is always 0. The ASCII code does not specify the value of the parity bit, and any value can be used. Different computers

may therefore use the ASCII code with even parity, odd parity, or no parity.

The code of the control character *DEL* is all ones (except the parity which is, as usual, unspecified).This is a tradition from the old days of computing (and also from telegraphy), when paper tape was an important medium for input/output. When punching information on a paper tape, whenever the user noticed an error, they would delete the bad character by pressing the DEL key on the keyboard. This worked by backspacing the tape and punching a frame of all 1's on top of the bad character. When reading the tape, the reader would simply skip any frame of all 1's.

0.7.1 Extended Binary Coded Decimal Interchange Code

A related code is the EBCDIC (Extended Binary Coded Decimal Interchange Code). This code was used on IBM computers and may still be used (as an optional alternative to ASCII) by some old IBM personal computers. EBCDIC is an 8-bit code, with room for up to 256 characters. However, it assigns codes to only 107 characters, and there are quite a few unassigned codes. The term BCD (binary coded decimal) refers to the binary codes of the 10 decimal digits. EBCDIC was developed by IBM, in the late 1950s, for its 360 computers. However, to increase compatibility, the 360 later received hardware that enabled it to also use the ASCII code. Because of the influence of IBM, some of the computers designed in the 1960s and 70s also use the EBCDIC code. Today, however, the ASCII code is a *de-facto* standard (with *unicode* catching on).

We are used to decimal numbers; computers find it easy to deal with binary numbers. Conversions are therefore necessary, and can easily be done by the computer. However, sometimes it is preferable to avoid number conversions. This is true in an application where many numbers have to be input, stored in memory, and output, with very little processing done between the input and the output. Perhaps a good example is inventory control. Imagine a large warehouse where thousands or even tens of thousands of items are stored. In a typical inventory control application, a record has to be input from a master file for each item, stored in memory, updated, perhaps printed, and finally written on a new master file. The updating usually involves a few simple operations such as incrementing or decrementing the number of units on hand. In such a case it may be better not to convert all the input to binary, which also saves conversion of the output from binary. Such a situation is typical in data processing applications, which is why computers designed specifically for

such applications support BCD numbers in hardware.

The EBCDIC control characters were developed to support punched card equipment and simple line printers. Missing are all the ASCII control characters used for telecommunications and for driving high-speed disk drives.

The idea in a BCD (binary coded decimal) number is to store the decimal digits of a number in memory, rather than converting the entire number to binary (integer or floating point). Since memory can only contain bits, each decimal digit has to be converted to bits, but this is a very simple process. As an example, consider the decimal number -8190. Converting this number to binary integer is time consuming (try it!) and the result is 13 bits long (14, including the sign bit). On the other hand, converting each decimal digit is quick and easy. It yields 1000 0001 1001 0000. Each decimal digit is represented as a group of four bits, since the largest digit (=9) requires four bits.

Practice Exercises

1. Express the decimal 0.5, -0.123 as signed 6-bit fractions.

2. What is the maximum representation error, e, if only 8 significant bits after decimal point are used?

3. Assuming a 6-bit exponent, 9-bit normalized fractional mantissa, and exponent is represented in biased format, add the number below:A = 0 100001 111111110, B=0 011111 0010110101. Assume an implicit 1 to the left of mantissa.

4. Assuming all numbers are in 2's complement representation, which of the following numbers is divisible by 11111011?
   (A) 11100111 (B) 11100100 (C) 11010111 (D) 11011011

5. The hexadecimal representation of 657_8 is:
   (a) 1AF (b) D78 (c) D71 (d) 32F

6. What is answer for $(1 + 1 \times 10^{20}) - (1 \times 10^{20})$? Justify.

7. What is answer for $1 + (1 \times 10^{20} - 1 \times 10^{20})$? Justify.

8. How the rounding/truncation is carried out by the CPU?

9. How overflow can be detected, if the carry bit is not used? I.e., decide it only based on the values of A, B, $C = A + B$ and $C = A - B$. Assume that two's complement is used for negative numbers.

10. The range of integers that can be represented by an n bit 2's complement number system is:
    (a) -2^{n-1} to $(2^{n-1}-1)$ (b) $-(2^{n-1}-1)$ to $(2^{n-1}-1)$
    (c) -2^{n-1} to 2^{n-1} (d) $-(2^{n-1}+1)$ to $(2^{n-1}-1)$

11. The following is a scheme for floating point number representation using 16 bits.

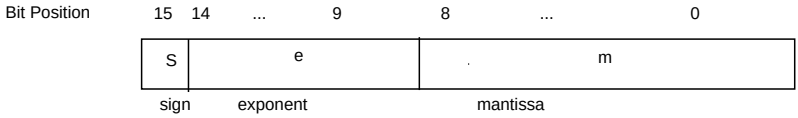

Figure 4: Floating point format

Let s, e, and m be the numbers represented in binary in the sign, exponent, and mantissa fields, respectively. Then the floating point number represented is: $(-1)^s(1 + m \times 2^{-9})2^{e-31}$, if the number $\neq 111111$, and 0 otherwise.

What is the maximum difference between two successive real numbers representable in this system?
(A) 2^{-40} (B) 2^{-9} (C) 2^{22} (D) 2^{31}

12. What are the values of expressions $\infty/0, 0/\infty, \infty/\infty$. Justify your answer.