There is no end carry

Answer is negative 59282 = 10's complement of 40718

Since we are dealing with unsigned numbers, there is really no way to get an unsigned result for the second example. When working with paper and pencil, we recognize that the answer must be changed to a signed negative number. When subtracting with complements, the negative answer is recognized by the absence of the end carry and the complemented result.

Subtraction with complements is done with binary numbers in a similar manner using the same procedure outlined above. Using the two binary numbers $X = 1010100$ and $Y = 1000011$, we perform the subtraction $X - Y$ and $Y - X$ using 2's complements:

$$\begin{aligned} X &= \quad 1010100 \\ \text{2's complement of } Y &= \quad +0111101 \\ \text{Sum} &= \quad 10010001 \\ \text{Discard end carry } 2^7 &= -10000000 \\ \text{Answer: } X - Y &= \quad 0010001 \end{aligned}$$

$$\begin{aligned} Y &= \quad +000011 \\ \text{2's complement of } X &= \quad +0101100 \\ \text{Sum} &= \quad 1101111 \end{aligned}$$

There is no end carry

Answer is negative 0010001 = 2's complement of 1101111

3-3 Fixed-Point Representation

Positive integers, including zero, can be represented as unsigned numbers. However, to represent negative integers, we need a notation for negative values. In ordinary arithmetic, a negative number is indicated by a minus sign and a positive number by a plus sign. Because of hardware limitations, computers must represent everything with l's and 0's, including the sign of a number. As a consequence, it is customary to represent the sign with a bit placed in the leftmost position of the number. The convention is to make the sign bit equal to 0 for positive and to 1 for negative.

binary point

In addition to the sign, a number may have a binary (or decimal) point. The position of the binary point is needed to represent fractions, integers, or mixed integer-fraction numbers. The representation of the binary point in a register is complicated by the fact that it is characterized by a position in the register. There are two ways of specifying the position of the binary point in a register: by giving it a fixed position or by employing a floating-point representation. The fixed-point method assumes that the binary point is

always fixed in one position. The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right of the register to make the stored number an integer. In either case, the binary point is not actually present, but its presence is assumed from the fact that the number stored in the register is treated as a fraction or as an integer. The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. Floating-point representation is discussed further in the next section.

Integer Representation

signed numbers

When an integer binary number is positive, the sign is represented by 0 and the magnitude by a positive binary number. When the number is negative, the sign is represented by 1 but the rest of the number may be represented in one of three possible ways:

1. Signed-magnitude representation
2. Signed-1's complement representation
3. Signed 2's complement representation

The signed-magnitude representation of a negative number consists of the magnitude and a negative sign. In the other two representations, the negative number is represented in either the l's or 2's complement of its positive value. As an example, consider the signed number 14 stored in an 8-bit register. +14 is represented by a sign bit of 0 in the leftmost position followed by the binary equivalent of 14: 00001110. Note that each of the eight bits of the register must have a value and therefore 0's must be inserted in the most significant positions following the sign bit. Although there is only one way to represent +14, there are three different ways to represent −14 with eight bits.

In signed-magnitude representation	1 0001110
In signed-1's complement representation	1 1110001
In signed-2's complement representation	1 1110010

The signed-magnitude representation of −14 is obtained from +14 by complementing only the sign bit. The signed-1's complement representation of −14 is obtained by complementing all the bits of +14, including the sign bit. The signed-2's complement representation is obtained by taking the 2's complement of the positive number, including its sign bit.

The signed-magnitude system is used in ordinary arithmetic but is awkward when employed in computer arithmetic. Therefore, the signed-complement is normally used. The l's complement imposes difficulties because

it has two representations of 0 (+0 and −0). It is seldom used for arithmetic operations except in some older computers. The l's complement is useful as a logical operation since the change of 1 to 0 or 0 to 1 is equivalent to a logical complement operation. The following discussion of signed binary arithmetic deals exclusively with the signed-2's complement representation of negative numbers.

Arithmetic Addition

The addition of two numbers in the signed-magnitude system follows the rules of ordinary arithmetic. If the signs are the same, we add the two magnitudes and give the sum the common sign. If the signs are different, we subtract the smaller magnitude from the larger and give the result the sign of the larger magnitude. For example, (+25) + (−37) = −(37 − 25) = −12 and is done by subtracting the smaller magnitude 25 from the larger magnitude 37 and using the sign of 37 for the sign of the result. This is a process that requires the comparison of the signs and the magnitudes and then performing either addition or subtraction. (The procedure for adding binary numbers in signed-magnitude representation is described in Sec. 10-2.) By contrast, the rule for adding numbers in the signed-2's complement system does not require a comparison or subtraction, only addition and complementation. The procedure is very simple and can be stated as follows: Add the two numbers, including their sign bits, and discard any carry out of the sign (leftmost) bit position. Numerical examples for addition are shown below. Note that negative numbers must initially be in 2's complement and that if the sum obtained after the addition is negative, it is in 2's complement form.

2's complement addition

+ 6	00000110	−6	11111010
+13	00001101	+13	00001101
+19	00010011	+7	00000111
+ 6	00000110	−6	11111010
−13	11110011	−13	11110011
−7	11111001	−19	11101101

In each of the four cases, the operation performed is always addition, including the sign bits. Any carry out of the sign bit position is discarded, and negative results are automatically in 2's complement form.

The complement form of representing negative numbers is unfamiliar to people used to the signed-magnitude system. To determine the value of a negative number when in signed-2's complement, it is necessary to convert it to a positive number to place it in a more familiar form. For example, the signed binary number 11111001 is negative because the leftmost bit is 1. Its 2's complement is 00000111, which is the binary equivalent of +7. We therefore recognize the original negative number to be equal to −7.

Arithmetic Subtraction

2's complement subtraction

Subtraction of two signed binary numbers when negative numbers are in 2's complement form is very simple and can be stated as follows: Take the 2's complement of the subtrahend (including the sign bit) and add it to the minuend (including the sign bit). A carry out of the sign bit position is discarded.

This procedure stems from the fact that a subtraction operation can be changed to an addition operation if the sign of the subtrahend is changed .This is demonstrated by the following relationship:

$$(\pm A) - (+B) = (\pm A) + (-B)$$
$$(\pm A) - (-B) = (\pm A) + (+B)$$

But changing a positive number to a negative number is easily done by taking its 2's complement. The reverse is also true because the complement of a negative number in complement form produces the equivalent positive number. Consider the subtraction of $(-6) - (-13) = +7$. In binary with eight bits this is written as 11111010 − 11110011. The subtraction is changed to addition by taking the 2's complement of the subtrahend (-13) to give $(+13)$. In binary this is 11111010 + 00001101 = 100000111. Removing the end carry, we obtain the correct answer 00000111 $(+7)$.

It is worth noting that binary numbers in the signed-2's complement system are added and subtracted by the same basic addition and subtraction rules as unsigned numbers. Therefore, computers need only one common hardware circuit to handle both types of arithmetic. The user or programmer must interpret the results of such addition or subtraction differently depending on whether it is assumed that the numbers are signed or unsigned.

Overflow

overflow

When two numbers of n digits each are added and the sum occupies $n + 1$ digits, we say that an overflow occurred. When the addition is performed with paper and pencil, an overflow is not a problem since there is no limit to the width of the page to write down the sum. An overflow is a problem in digital computers because the width of registers is finite. A result that contains $n + 1$ bits cannot be accommodated in a register with a standard length of n bits. For this reason, many computers detect the occurrence of an overflow, and when it occurs, a corresponding flip-flop is set which can then be checked by the user.

The detection of an overflow after the addition of two binary numbers depends on whether the numbers are considered to be signed or unsigned. When two unsigned numbers are added, an overflow is detected from the end carry out of the most significant position. In the case of signed numbers, the leftmost bit always represents the sign, and negative numbers are in 2's complement form. When two signed numbers are added, the

sign bit is treated as part of the number and the end carry does not indicate an overflow.

An overflow cannot occur after an addition if one number is positive and the other is negative, since adding a positive number to a negative number produces a result that is smaller than the larger of the two original numbers. An overflow may occur if the two numbers added are both positive or both negative. To see how this can happen, consider the following example. Two signed binary numbers, +70 and +80, are stored in two 8-bit registers. The range of numbers that each register can accommodate is from binary +127 to binary −128. Since the sum of the two numbers is +150, it exceeds the capacity of the 8-bit register. This is true if the numbers are both positive or both negative. The two additions in binary are shown below together with the last two carries.

carries:	0	1	carries:	1	0
+70	0	1000110	−70	1	0111010
+80	0	1010000	−80	1	0110000
+150	1	0010110	−150	0	1101010

Note that the 8-bit result that should have been positive has a negative sign bit and the 8-bit result that should have been negative has a positive sign bit. If, however, the carry out of the sign bit position is taken as the sign bit of the result, the 9-bit answer so obtained will be correct. Since the answer cannot be accommodated within 8 bits, we say that an overflow occurred.

overflow detection

An overflow condition can be detected by observing the carry into the sign bit position and the carry out of the sign bit position. If these two carries are not equal, an overflow condition is produced. This is indicated in the examples where the two carries are explicitly shown. If the two carries are applied to an exclusive-OR gate, an overflow will be detected when the output of the gate is equal to 1.

Decimal Fixed-Point Representation

The representation of decimal numbers in registers is a function of the binary code used to represent a decimal digit. A 4-bit decimal code requires four flip-flops for each decimal digit. The representation of 4385 in BCD requires 16 flip-flops, four flip-flops for each digit. The number will be represented in a register with 16 flip-flops as follows:

0100 0011 1000 0101

By representing numbers in decimal we are wasting a considerable amount of storage space since the number of bits needed to store a decimal number in a binary code is greater than the number of bits needed for its equivalent binary representation. Also, the circuits required to perform decimal

arithmetic are more complex. However, there are some advantages in the use of decimal representation because computer input and output data are generated by people who use the decimal system. Some applications, such as business data processing, require small amounts of arithmetic computations compared to the amount required for input and output of decimal data. For this reason, some computers and all electronic calculators perform arithmetic operations directly with the decimal data (in a binary code) and thus eliminate the need for conversion to binary and back to decimal. Some computer systems have hardware for arithmetic calculations with both binary and decimal data.

The representation of signed decimal numbers in BCD is similar to the representation of signed numbers in binary. We can either use the familiar signed-magnitude system or the signed-complement system. The sign of a decimal number is usually represented with four bits to conform with the 4-bit code of the decimal digits. It is customary to designate a plus with four 0's and a minus with the BCD equivalent of 9, which is 1001.

The signed-magnitude system is difficult to use with computers. The signed-complement system can be either the 9's or the 10's complement, but the 10's complement is the one most often used. To obtain the 10's complement of a BCD number, we first take the 9's complement and then add one to the least significant digit. The 9's complement is calculated from the subtraction of each digit from 9.

The procedures developed for the signed-2's complement system apply also to the signed-10's complement system for decimal numbers. Addition is done by adding all digits, including the sign digit, and discarding the end carry. Obviously, this assumes that all negative numbers are in 10's complement form. Consider the addition $(+375) + (-240) = +135$ done in the signed-10's complement system.

$$\begin{array}{rl} 0\ 375 & (0000\ 0011\ 0111\ 0101)_{\text{BCD}} \\ +\ 9\ 760 & (1001\ 0111\ 0110\ 0000)_{\text{BCD}} \\ \hline 0\ 135 & (0000\ 0001\ 0011\ 0101)_{\text{BCD}} \end{array}$$

The 9 in the leftmost position of the second number indicates that the number is negative. 9760 is the 10's complement of 0240. The two numbers are added and the end carry is discarded to obtain +135. Of course, the decimal numbers inside the computer must be in BCD, including the sign digits. The addition is done with BCD adders (see Fig. 10-18).

The subtraction of decimal numbers either unsigned or in the signed-10's complement system is the same as in the binary case. Take the 10's complement of the subtrahend and add it to the minuend. Many computers have special hardware to perform arithmetic calculations directly with decimal numbers in BCD. The user of the computer can specify by programmed instructions that the arithmetic operations be performed with decimal numbers directly without having to convert them to binary.

3-4 Floating-Point Representation

mantissa
exponent

The floating-point representation of a number has two parts. The first part represents a signed, fixed-point number called the mantissa. The second part designates the position of the decimal (or binary) point and is called the exponent. The fixed-point mantissa may be a fraction or an integer. For example, the decimal number +6132.789 is represented in floating-point with a fraction and an exponent as follows:

Fraction	*Exponent*
+0.6132789	+04

The value of the exponent indicates that the actual position of the decimal point is four positions to the right of the indicated decimal point in the fraction. This representation is equivalent to the scientific notation $+0.6132789 \times 10^{+4}$.

Floating-point is always interpreted to represent a number in the following form:

$$m \times r^e$$

Only the mantissa m and the exponent e are physically represented in the register (including their signs). The radix r and the radix-point position of the mantissa are always assumed. The circuits that manipulate the floating-point numbers in registers conform with these two assumptions in order to provide the correct computational results.

fraction

A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent. For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

Fraction	*Exponent*
01001110	000100

The fraction has a 0 in the leftmost position to denote positive. The binary point of the fraction follows the sign bit but is not shown in the register. The exponent has the equivalent binary number +4. The floating-point number is equivalent to

$$m \times 2^e = +(.1001110)_2 \times 2^{+4}$$

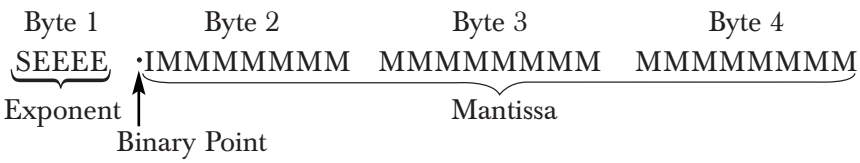
normalization

A floating-point number is said to be *normalized* if the most significant digit of the mantissa is nonzero. For example, the decimal number 350 is normalized but 00035 is not. Regardless of where the position of the radix point is assumed to be in the mantissa, the number is normalized only if its leftmost digit is nonzero. For example, the 8-bit binary number 00011010 is not normalized because of the three leading 0's. The number can be normalized by shifting it

three positions to the left and discarding the leading 0's to obtain 11010000. The three shifts multiply the number by $2^3 = 8$. To keep the same value for the floating-point number, the exponent must be subtracted by 3. Normalized numbers provide the maximum possible precision for the floating-point number. A zero cannot be normalized because it does not have a nonzero digit. It is usually represented in floating-point by all 0's in the mantissa and exponent.

Two main standard forms of floating-point numbers are from the following organizations that decide standards: ANSI (American National Standards Institute) and IEEE (Institute of Electrical and Electronic Engineers). The ANSI 32-bit floating-point numbers in byte format with examples are given below:

Byte Format:



S = Sign of Mantissa, E = Exponent Bits in 2's complement, M = Mantissa Bits

Examples:

$$\begin{aligned} 13 = 1101 &= 0.1101 \times 2^4 \\ &= 00000100\ 11010000\ 00000000\ 00000000 \\ -17 = -10001 &= -0.10001 \times 2^5 \\ &= 10000101\ 10001000\ 00000000\ 00000000 \\ -0.125 = -0.001 &= -.1 \times 2^{-2} \\ &= 11111110\ 10000000\ 00000000\ 00000000 \end{aligned}$$

Arithmetic operations with floating-point numbers are more complicated than arithmetic operations with fixed-point numbers and their execution takes longer and requires more complex hardware. However, floating-point representation is a must for scientific computations because of the scaling problems involved with fixed-point computations. Many computers and all electronic calculators have the built-in capability of performing floating-point arithmetic operations. Computers that do not have hardware for floating-point computations have a set of subroutines to help the user program scientific problems with floating-point numbers. Arithmetic operations with floating-point numbers are discussed in Sec. 10-5.

3-5 Other Binary Codes

In previous sections we introduced the most common types of binary-coded data found in digital computers. Other binary codes for decimal numbers and