CHAPTER 2

Number Systems

Inside computers, information is encoded as patterns of bits because it is easy to construct electronic circuits that exhibit the two alternative states, 0 and 1. The meaning of bits depends on the interpretation. Since the information is processed by the computer essentially in some numerical form, we explore the ways how a computer represent numeric data internally in this chapter.

2.1 Positional Number Systems

A *number system* is a system of representing numbers. It is defined by a set of basic symbols called *digits* or *numerals*, and the ways in which the digits can be combined to represent the numbers in the system. A number system can represent integers, fractions, or mixed numbers. A *mixed number* has two parts: an integer part that tells you the whole and a fraction part that is less than one whole. A *radix point* separates the integer part and the fraction part.

Since the decimal number system is the most familiar number system and used in our everyday life, our discussion begins with it. The *decimal number system* is a *positional number system.* In a positional number system, a number is represented by a string of digits. The value of each digit in the string is determined by the position it occupies in the number. That is, each digit position has a weight associated with it. The rightmost position in the number has the lowest weight. The leftmost digit in the number is called the *most significant digit* (MSD) and the rightmost digit is called the *least significant digit* (LSD). Other examples of commonly used positional number systems include binary, octal, and hexadecimal number systems.

In the decimal number system, there are 10 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The decimal number of the form $d_{p-1}d_{p-2}...d_1d_0.d_{-1}d_{-2}...d_{-n}$ has the

Name	Base	Digits
Binary	2	0, 1
Octal	8	0, 1, 2, 3, 4, 5, 6, 7
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Table 2.1: Commonly used number systems with their bases and digit sets.

value,

$$\sum_{i=-n}^{p-1} d_i \cdot 10^i$$

All the weights in the decimal system are powers of the number 10. The integer and fraction parts are separated by the . symbol, the *decimal point*. The fraction part is denoted by a sequence of digits whose weights are negative powers of 10. The decimal number system is also known as the *base-10* number system because the weight of each position in the number is a power of 10. For example, the decimal number 754.82 can be decomposed as follows:

$$754.82 = 7 \times 10^2 + 5 \times 10^1 + 4 \times 10^0 + 8 \times 10^{-1} + 2 \times 10^{-2}$$

When we replace the base 10 with some other whole number r, called the *base* or *radix*, then we have *base-r number system*. This number system requires r distinct digits, and the weight in position i is r^i. Thus, the base-r number of the form $x_{p-1}x_{p-2}...x_1x_0.x_{-1}x_{-2}...x_{-n}$ has the value,

$$\sum_{i=-n}^{p-1} x_i \cdot r^i$$

where each x^i comes from the set of r distinct digits. Traditionally, the first r decimal digits serve as the digits for the base-r systems when $r \leq 10$. When $r > 10$, the first $r - 10$ uppercase letters of the alphabet are used to provide additional symbols.

For example, the three-digit number 101 denotes five in the binary number system whereas it denotes one hundred and one in the decimal system. When the base of a base-r number N is unclear from the context, we append a subscript r to it. For example, 101_2 represent a binary number 101. Table 2.1 shows the bases and digit sets for some commonly used number systems.

2.2 Scientific Notation

Scientific notation is a scheme of representing decimal numbers that are very small or vary large. In scientific notation, a number is represented in the form:

$$x \times 10^y$$

Copyright ©2012 by Jaejin Lee

Where x is called the *coefficient* (also called the *significand* or *mantissa*) and is any real number, and y is called the *exponent* and must be an integer. For example, -320000000 can be written as -32.0×10^7 in scientific notation.

Since there are many ways to represent a number in the form of $x \times 10^y$, we adopt the convention of making the significand, x, always in the range: $1 \leq |x| < 10$. This notation is often referred to as *normalized scientific notation.* Note that we can not express zero with the normalized scientific notation.

In a positional number system, the *significant digits* of a number are those digits whose removal changes the numerical value associated with the number. A number's *precision* or *accuracy* is the number of significant digits it contains. *Leading zeroes* of a number are any consecutive zeroes that appear in the leftmost positions of the number's representation. On the other hand, *trailing zeroes* are any consecutive zeroes in the number's representation after which no other digits follow. Leading zeroes are insignificant and do not affect the numerical value. Similarly, trailing zeroes that appears to the right of a radix point do not affect the value. They are merely placeholders to indicate the size of the representation. For example, the removal of leading and trailing zeroes in 0003.1400_{10} does not affect the value of the representation.

Base-r numbers expressed with a fixed number of digits often have an *implicit* radix point at some *fixed* position. For example, if the number is an integer, the radix point is immediately to the right of its LSD. If it is a fraction, the radix point is immediately to the left of its MSD. These numbers are referred to as *fixed-point numbers.* In contrast, *floating-point numbers* have a radix point that can be placed anywhere relative to their significant digits: the radix point can *float.* The position of the radix point is indicated separately and encoded in the number's representation. Scientific notation is closely related to the floating-point numbers. We will describe later how the computers express floating-point numbers internally.

2.3 Binary Number System

Since it was easier to build electronic circuits that distinguish between two different values than more than two values, computers use *Boolean logic*, which is a two-valued logical system, to abstract away the details of electronic circuits. Consequently, the binary number system are directly related to the Boolean logic used in the computer, and the computer internally represents numeric data in a binary form.

As shown in Table 2.1, the binary number system uses 0 and 1 as its digits. The general form of a binary number is $b_{p-1}b_{p-2}...b_1b_0.b_{-1}b_{-2}...b_{-n}$, and it has the following interpretation,

$$\sum_{i=-n}^{p-1} b_i \cdot 2^i$$

For example,

$$10011_2 = 1 \cdot 2^4 + 1 \cdot 2^1 + 1 \cdot 2^0 = 19$$

Copyright ©2012 by Jaejin Lee

Binary	Octal	Hexadecimal	Decimal
0000	00	0	0
0001	01	1	1
0010	02	2	2
0011	03	3	3
0100	04	4	4
0101	05	5	5
0110	06	6	6
0111	07	7	7
1000	10	8	8
1001	11	9	9
1010	12	A	10
1011	13	B	11
1100	14	C	12
1101	15	D	13
1110	16	E	14
1111	17	F	15

Table 2.2: The correspondence between 4-bit binary, octal, hexadecimal, and decimal numbers.

$$101.001_2 = 1 \cdot 2^2 + 1 \cdot 2^0 + 1 \cdot 2^{-3} = 5.125$$

Similar to the decimal number system, the leftmost bit b_{p-1} is called the *most significant bit* (MSB) and the rightmost bit b_{-n} is called the *least significant bit* (LSB). The radix point in the binary number system is referred to as the *binary point*.

The octal number system is useful for representing multi-bit binary numbers. Since eight is a power of two ($8 = 2^3$), three bits in a binary number can be uniquely represented with a single octal digit. For example,

$$100011001110_2 = 100\ 011\ 001\ 110_2 = 4316_8$$

$$10.1011001011_2 = 010\ .\ 101\ 100\ 101\ 100_2 = 2.5454_8$$

Similarly, the hexadecimal number system is also useful for representing multi-bit binary numbers. Each group of four bits in a binary number can be uniquely represented by a single hexadecimal digit. For example,

$$100011001110_2 = 1000\ 1100\ 1110_2 = 8CE_{16}$$

$$10.1011001011_2 = 0010\ .\ 1011\ 0010\ 1100_2 = 2.B2C_{16}$$

Table 2.2 shows the correspondence between 4-bit binary, octal, hexadecimal, and decimal numbers.

Copyright ©2012 by Jaejin Lee

Let N be the decimal number and $b_{n-1}b_{n-2} \cdots b_0$ be the binary number after the conversion.

1. Set i to 0.

2. Divide N by 2 and obtain a quotient and a remainder.

3. Set b_i to the remainder and N to the quotient.

4. If N is not zero, set i to $i+1$ and go to step 2.

Figure 2.1: A decimal to binary conversion algorithm for an integer.

i	N	b_i	
0	$108/2 = 54$	0	(LSB)
1	$54/2 = 27$	0	
2	$27/2 = 13$	1	
3	$13/2 = 6$	1	
4	$6/2 = 3$	0	
5	$3/2 = 1$	1	
6	$1/2 = 0$	1	(MSB)

Figure 2.2: Converting 108_{10} to binary.

A binary number system using n bits can represent 2^n different numbers. When such a fixed precision binary number is used to represent only positive values, it is called an *unsigned number*. It is also possible to encode *signed numbers* (positive numbers and negative numbers) in binary. There are several ways to represent the signed numbers in binary. One way of representing a negative number is setting the MSB to 1 and using the remaining bits to represent the value. In this case, the MSB is referred to as the *sign bit*. However, to keep the computer hardware implementation as simple as possible, almost all today's computers internally use *twos complement representation*.

2.4 Number System Conversion

Since humans typically use decimal numbers, but computers use binary numbers internally, it is important to know how to convert decimal numbers to binary numbers and *vice versa*. An integer N can be converted from decimal form to binary form by repeatedly dividing N by two and using the remainders. Figure 2.1 shows an algorithm for such a conversion. For example, a positive integer 108_{10} is converted to the binary number 1101100_2. The conversion process that follows the algorithm described in Figure 2.1 is illustrated in Figure 2.2.

Copyright ©2012 by Jaejin Lee

Let ***N*** be the decimal fraction and $b_{-1}b_{-2} \cdots b_{-n}$ be the binary number after the conversion.

1. Set ***i*** to 1.

2. Multiply ***N*** by 2 and set ***N*** to the result.

3. If ***N*** is less than 1, set ***b-i*** to 0.

4. Otherwise, set ***b-i*** to 1 and ***N*** to ***N***-1.

5. If ***i*** is less than ***n***, set ***i*** to ***i+1*** and go to step 2.

Figure 2.3: A decimal to binary conversion algorithm for fractions.

i	N	b_{-i}
1	$0.735 \times 2 = 1.47$	1
2	$0.47 \times 2 = 0.94$	0
3	$0.94 \times 2 = 1.88$	1
4	$0.88 \times 2 = 1.76$	1

Figure 2.4: Converting 0.731_{10} to a 4-bit binary fraction.

For a mixed number, we convert its integer part and fraction part to binary form separately. Then, we combine the results. An algorithm for converting a decimal fraction to binary form is shown in Figure 2.3. To obtain the first n bits of a binary fraction using the algorithm, it is required to perform n multiplications by two in general. For example, the process of converting a decimal fraction 0.731_{10} to a 4-bit binary fraction 0.1011_2 is illustrated in Figure 2.4. After combining the results from the integer conversion and the fraction conversion, a mixed number 108.731_{10} in the above examples is converted to a binary number 1101100.1011_2.

However, the 4-bit binary fraction obtained in the process of Figure 2.4 is inexact. So far, we assumed that there are as many bits available as required to represent the number, and we did not consider the size of the representation. When the number of bits used is specified to represent a number, it determines the range of possible numbers that can be represented. For example, 8 bits can represent $256 = 2^8$ distinct numeric values. *Word* is a group of bits that are handled as a unit by the computer. The number of bits in a word is an important characteristic of a computer architecture. The precision of expressing a numerical quantity strongly depends on the word size (i.e., the number of bits in a word). If the word size is only four bits, then we can not express the result of the above example more precisely.

A *dyadic fraction* is a rational number whose denominator is a power of two;

Copyright ©2012 by Jaejin Lee

i	N	b_i	
0	$108/8 = 13$	4	(LSB)
1	$13/8 = 1$	5	
2	$1/8 = 0$	1	(MSB)

i	N	b_{-i}
1	$0.735 \times 8 = 5.88$	5
2	$0.88 \times 8 = 7.04$	7
3	$0.04 \times 8 = 0.32$	0
4	$0.32 \times 8 = 2.56$	2

Figure 2.5: Converting 108.731_{10} to an octal number.

i	N	b_i	
0	$108/16 = 6$	$12(C)$	(LSB)
1	$6/16 = 0$	6	(MSB)

i	N	b_{-i}
1	$0.735 \times 16 = 11.76$	$11(B)$
2	$0.76 \times 16 = 12.16$	$12(C)$
3	$0.16 \times 16 = 2.56$	2
4	$0.56 \times 16 = 8.96$	8

Figure 2.6: Converting 108.731_{10} to a hexadecimal number.

for example, 1/2 or 3/16, but not 1/3. Dyadic decimal fractions convert to finite binary fractions and are represented in full precision if the word size is greater than or equal to the length of the binary representation. Non-dyadic decimal fractions convert to infinite binary fractions (a repeating sequence of the same bit pattern). *Truncation* (also known as *chopping*) is the process of discarding any unwanted digits of a number. *Rounding* a number replaces the number with another number that is as close to the original number as possible.

Decimal to octal and decimal to hexadecimal conversions are similar to the decimal to binary conversion. An integer can be converted from decimal form to octal (or hexadecimal) form by repeatedly dividing it by eight (or sixteen) and using the remainders. A decimal fraction converts an octal (or hexadecimal) fraction with n digits after performing n successive multiplications by eight (or sixteen). For example, the conversion process of a mixed number, 108.731_{10}, to octal (154.5702_8) and hexadecimal ($6C.BC28_{16}$) are illustrated in Figure 2.5 and Figure 2.6, respectively.

Another way to perform decimal to octal or decimal to hexadecimal conversion for a decimal number is that converting it to a binary number. Then, the binary number can be converted to an equivalent octal or hexadecimal number. For example, 108.731_{10} is converted to octal (154.5702_8) and hexadecimal ($6C.BC28_{16}$) in the following way:

$$\begin{aligned}108.731_{10} &= 001\ 101\ 100\ .\ 101\ 111\ 000\ 010_2\\ &= 154.5702_8\\ 108.731_{10} &= 0110\ 1100\ .\ 1011\ 1100\ 0010\ 1000_2\\ &= 6C.BC28_{16}\end{aligned}$$

Copyright ©2012 by Jaejin Lee

2.5 Unsigned Binary Arithmetic

Arithmetic operations performed on unsigned binary numbers are much like arithmetic in the decimal number system that we know very well. Addition, subtraction, multiplication, and division can be performed on binary numbers. A pair of unsigned binary numbers can be added bit by bit according to the same method used in decimal addition. Since computers use different standard to encode a mixed number (typically using the IEEE 754 standard), we will restrict our discussion to unsigned whole numbers.

For example, a pair of unsigned 4-bit binary numbers 1001 and 0101 can be added as follows:

carry	0	0	0	1	
		1	0	0	1
+		0	1	0	1
	0	1	1	1	0

When adding two 1s in the same column produces 10_2, 1 (called *carry*) will have to be added to the left column of more significant bits. When the result of addition exceeds one, we carry the excess amount to the next position using a carry. The position has a weight that is higher by a factor equal to two.

An *overflow* occurs when a computation produces a value that falls outside the range of values that can be represented. For example, adding two unsigned 4-bit binary numbers 1011 and 0111 produces a 5-bit value 10010, and it is an overflow:

carry	1	1	1	1	
		1	0	1	1
+		0	1	1	1
	1	0	0	1	0

The carry bit from the MSB tell us that an overflow occurred in the addition.

Unsigned binary subtraction works in much the same way as decimal subtraction. For example, 0101 can be subtracted from 1001 produces 0100 as follows:

borrow	0	1	0	0	
		1	0	0	1
−		0	1	0	1
	0	0	1	0	0

Subtracting a 1 from a 0 in a column produces 1 by borrowing 1 (called *borrow*) from the left column of more significant bits. When the result of a bit by bit subtraction in a column is less than 0, we borrow 1 from the column on the left subtracting it from the next higher positional value.

Unsigned binary multiplication is also similar to the decimal multiplication that adds up partial products to produce the final result. For example, two

Copyright ©2012 by Jaejin Lee

Positive		Negative	
Ones' complement	Decimal	Ones' complement	Decimal
0000	0	1111	-0
0001	1	1110	-1
0010	2	1101	-2
0011	3	1100	-3
0100	4	1011	-4
0101	5	1010	-5
0110	6	1001	-6
0111	7	1000	-7

Table 2.3: 4-bit ones' complement representation.

4-bit unsigned binary numbers 1001 and 0101 are multiplied as follows:

$$
\begin{array}{cccccccc}
 & & & & 1 & 0 & 0 & 1 \\
\times & & & & 0 & 1 & 0 & 1 \\
\hline
 & & & & 1 & 0 & 0 & 1 \\
 & & & 0 & 0 & 0 & 0 & \\
 & & 1 & 0 & 0 & 1 & & \\
 & 0 & 0 & 0 & 0 & & & \\
\hline
 & 0 & 1 & 0 & 1 & 1 & 0 & 1
\end{array}
$$

Note that the result can not be represented with 4 bits. In general, an n-bit binary multiplication requires $2 \cdot n$ bits to represent the result.

Unsigned binary division is again similar to the decimal division. For example, the process of dividing 11101 by 0100 produces a quotient 111 and a remainder 1.

$$
\begin{array}{rcccccc}
 & & & & 1 & 1 & 1 \\
\cline{3-7}
100 &) & 1 & 1 & 1 & 0 & 1 \\
 & & 1 & 0 & 0 & & \\
\cline{3-7}
 & & 0 & 1 & 1 & 0 & 1 \\
 & & & 1 & 0 & 0 & \\
\cline{3-7}
 & & & 0 & 1 & 0 & 1 \\
 & & & & 1 & 0 & 0 \\
\cline{3-7}
 & & & & 0 & 0 & 1
\end{array}
$$

2.6 Ones' Complement Representation

Ones' complement representation is one of the methods to represent a negative number. The ones' complement of a negative binary number is the bitwise inversion (flipping 0's for 1's and *vice-versa*) of its positive counterpart. For example, the ones complement representation of -6 (-0110_2) is 1001_2.

The maximum value that can be represented by the ones' complement representation with n bits is $2^{n-1}-1$ and the minimum is $-2^{n-1}-1$. Table 2.3 shows

Copyright ©2012 by Jaejin Lee

the numbers that can be represented in the ones' complement representation with 4 bits. The MSB is the sign bit. Unlike two's complement representation (explained later), it has two zeroes. For example, 0000_2 and 1111_2 are the two 4-bit ones' complement representations of zero.

Adding two binary numbers in the ones' complement representation is similar to the unsigned binary addition, but there is a condition called an *end-around carry* that does not occur in unsigned binary addition. Consider adding two binary numbers 0100 and 1110 in ones' complement representation:

carry	1	1	0	0		
		0	1	0	0	(4)
+		1	1	1	0	(−1)
		0	0	1	0	
					1	(add the end-around carry)
		0	0	1	1	(3)

For the addition in the ones' complement representation, we perform a unsigned binary addition. If there is a carry from the MSB (this is called an *end-around carry*), then add the carry back into the resulting sum to produce the final result.

Unlike the two's complement representation, the ones' complement representation is not popular in these days because of several issues like negative zero, end-around carry, etc.

2.7 Two's Complement Representation

The twos complement representation is the most common method of representing signed integers internally in computers. It simplifies the complexity of the ALU by allowing using the same circuit that is used to implement arithmetic operations for unsigned numbers. The major difference is the interpretation of the result produced by the arithmetic operation.

To convert a number to two's complement representation, the number is converted to its ones' complement first. Then one is added to the result to produce its two's complement. Another way of encoding a negative number to its twos complement is flipping all bits but the first least significant 1 and all the trailing 0s. For example, the twos complement representation of 6 (0110_2) is 1010_2.

The maximum value that can be represented by n-bit two's complement representation is $2^{n-1} - 1$ and the minimum is -2^{n-1}. Table 2.4 shows the numbers that can be represented in the two's complement representation with 4 bits. The MSB is the sign bit.

Two's complement addition is exactly the same as that of unsigned binary numbers. For example, adding 4 (0100_2) and 5 (1011_2) in 4-bit two's complement representation gives 1 (1111_2). Subtraction can be handled by converting it to addition:

$$x - y = x + (-y)$$

Copyright ©2012 by Jaejin Lee

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

Table 2.4: 4-bit two's complement representation.

1. Add the two operands including the sign bits.
2. If the carry into the MSB is not equal to the carry out of the MSB, an overflow has occurred.

Figure 2.7: Overflow detection in two's complement addition.

Overflow occurs if $n+1$ bits are required to contain the result from an n-bit addition or subtraction. If the sign bits were the same for both numbers and the sign of the result is different to them, an overflow has occurred. The process of detecting an overflow after adding two operands is shown in Figure 2.7.

For example, an overflow occurs in the following 4-bit two's complement addition:

$$\begin{array}{rccccccl} \text{carry} & 0 & 1 & 1 & 1 & & \\ & & 0 & 1 & 1 & 1 & (7) \\ + & & 0 & 1 & 1 & 1 & (7) \\ \hline & 0 & 1 & 1 & 1 & 0 & \end{array}$$

In this case, the carry into the MSB (1) is different to the carry (0) out of the MSB.

2.8 Shift Operations and Sign Extension

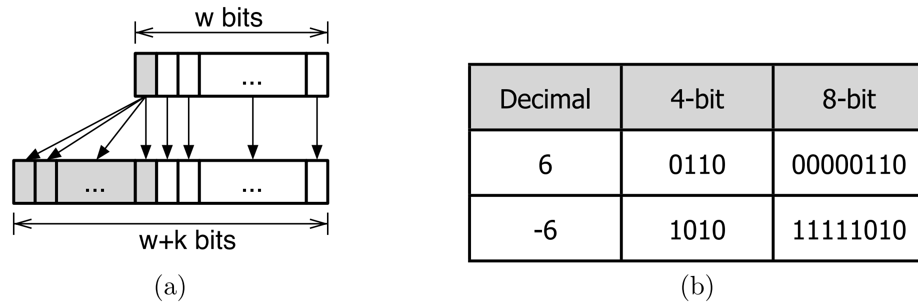
A shift operation shifts all of the bits in its operand. Every bit in its operand is moved a given number of positions to a specified direction. There two different types of *shift* operations: *logical shift* and *arithmetic shift*.

A logical shift operation moves bits to the left or right. The bits that fall off at the end of the word are discarded. The vacant positions in the opposite

Copyright ©2012 by Jaejin Lee

	$x >> 3$	$x << 3$
Logical shift	00010011	11101000
Arithmetic shift	11110011	11101000

Figure 2.8: The difference between 8-bit logical shift and arithmetic shift for $x = 10011101$.



Decimal	4-bit	8-bit
6	0110	00000110
-6	1010	11111010

(a) (b)

Figure 2.9: (a) The sign extension process. (b) The sign-bit repetition for -6.

end are filled with 0s. The arithmetic left shift is the same as the logical left shift. In the arithmetic right shift, the leftmost bits are filled with the sign bit of the original number. We denote a *left shift* operation with $<<$ and *right shift* operation with $>>$. Figure 2.8 shows an example of the difference between logical shift and arithmetic shift for $x = 10011101$.

Using shift operations, we can efficiently perform unsigned multiplication or division by powers of two. An n-bit logical left shift operation on unsigned integers is equivalent to multiplication by 2^n. For example,

$$00001011_2 \ (13) << 2 = 13 \times 2^2 = 00101100_2 \ (52)$$

An n-bit logical right shift is equivalent to division by 2^n, and we obtain the quotient of the division. For example,

$$00101101_2 \ (53) >> 2 = 53 \div 2^2 = 00001011_2 \ (13)$$

Using arithmetic shifts, we can efficiently perform multiplication or division of signed integers by powers of two. In two's complement representation, arithmetic shift operations extend the notion of the *floor* operation. The floor of an integer x (denoted by $\lfloor x \rfloor$) is defined by the greatest integer less than or equal to x. For example, $\lfloor 11/2 \rfloor = 5$ and $\lfloor -3/2 \rfloor = -2$. An arithmetic shift operation on an integer takes the floor of the result of multiplication or division. For example,

$$\begin{array}{rcll} 1101_2 \ \ (-4) << 1 & = & 1010_2 \ \ (-6) & \\ 1101_2 \ \ (-4) >> 1 & = & 1110_2 \ \ (-2) & \\ 1101_2 \ \ (-4) >> 2 & = & 1111_2 \ \ (-1) & \\ 1100_2 \ \ (-4) << 2 & = & 0000_2 \ \ (0) & \text{Overflow} \\ 0100_2 \ \ (4) << 1 & = & 1000_2 \ \ (-8) & \text{Overflow} \end{array}$$

Copyright ©2012 by Jaejin Lee

However, in two's complement representation, an arithmetic right shift is not equivalent to division by a power of two. For example, when you perform an arithmetic right shift on a 4-bit $-1 = 1111_2$, you still get $-1 = 1111_2$. The ANSI C99 standard does not specify the definition of the C language's right shift operator for negative values. The behavior of the right shift operator depends on the C compiler.

When converting an integer with w bits into the one with $w + k$ bits and the same value, we need to perform *sign extension* in two's complement representation. The MSB (the sign bit) must be repeated in all the k extra bits. Figure 2.9(a) shows this process and Figure 2.9(b).

Copyright ©2012 by Jaejin Lee