For 32 bit systems → 4 bytes

64bit systems → 8 bytes

Most of the data types encode signed values, unless prefixed by the keyword `unsigned` or using the specific unsigned declaration for fixed-size data types. The exception to this is data type `char`. Although most compilers and machines treat

x = 0x01234567

Big endian

	0x100	0x101	0x102	0x103	
...	01	23	45	67	...

from MSB to LSB

Little endian

	0x100	0x101	0x102	0x103	
...	67	45	23	01	...

from LSB to MSB

As our examples show, the best way to determine the effect of a bit-level expression is to expand the hexadecimal arguments to their binary representations, perform the operations in binary, and then convert back to hexadecimal.

```
1  void inplace_swap(int *x, int *y) {
2      *y = *x ^ *y;  /* Step 1 */
3      *x = *x ^ *y;  /* Step 2 */
4      *y = *x ^ *y;  /* Step 3 */
5  }
```

```
1  void reverse_array(int a[], int cnt) {
2      int first, last;
3      for (first = 0, last = cnt-1;
4           first <= last;
5           first++,last--)
6           inplace_swap(&a[first], &a[last]);
7  }
```

Bit Masking

One common use of bit-level operations is to implement *masking* operations, where a mask is a bit pattern that indicates a selected set of bits within a word. As an example, the mask `0xFF` (having ones for the least significant 8 bits) indicates the low-order byte of a word. The bit-level operation `x & 0xFF` yields a value consisting of the least significant byte of `x`, but with all other bytes set to 0. For example, with `x = 0x89ABCDEF`, the expression would yield `0x000000EF`. The expression `~0` will yield a mask of all ones, regardless of the size of the data representation. The same mask can be written `0xFFFFFFFF` when data type `int` is 32 bits, but it would not be as portable.

→ 0x000000FF

x = 0x89ABCDEF

~0 = 0xFFFFFFFF

expression `x << k` yields a value with bit representation $[x_{w-k-1}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0]$. That is, `x` is shifted k bits to the left, dropping off the k most significant bits and filling the right end with k zeros. The shift amount should be a value between 0 and $w-1$. Shift operations associate from left to right, so `x << j << k` is equivalent to `(x << j) << k`.

Logical. A logical right shift fills the left end with k zeros, giving a result $[0, \ldots, 0, x_{w-1}, x_{w-2}, \ldots x_k]$.

Arithmetic. An arithmetic right shift fills the left end with k repetitions of the most significant bit, giving a result $[x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots x_k]$.

Aside Shifting by k, for large values of k

For a data type consisting of w bits, what should be the effect of shifting by some value $k \geq w$? For example, what should be the effect of computing the following expressions, assuming data type `int` has $w = 32$:

```
int      lval = 0xFEDCBA98  << 32;
int      aval = 0xFEDCBA98  >> 36;
unsigned uval = 0xFEDCBA98u >> 40;
```

The C standards carefully avoid stating what should be done in such a case. On many machines, the shift instructions consider only the lower $\log_2 w$ bits of the shift amount when shifting a w-bit value, and so the shift amount is computed as $k \bmod w$. For example, with $w = 32$, the above three shifts would be computed as if they were by amounts 0, 4, and 8, respectively, giving results

```
lval    0xFEDCBA98
aval    0xFFEDCBA9
uval    0x00FEDCBA
```

This behavior is not guaranteed for C programs, however, and so shift amounts should be kept less than the word size.

Java, on the other hand, specifically requires that shift amounts should be computed in the modular fashion we have shown.

Aside Operator precedence issues with shift operations

It might be tempting to write the expression `1<<2 + 3<<4`, intending it to mean `(1<<2) + (3<<4)`. However, in C the former expression is equivalent to `1 << (2+3) << 4`, since addition (and subtraction) have higher precedence than shifts. The left-to-right associativity rule then causes this to be parenthesized as `(1 << (2+3)) << 4`, giving value 512, rather than the intended 52.

Getting the precedence wrong in C expressions is a common source of program errors, and often these are difficult to spot by inspection. When in doubt, put in parentheses!

PRINCIPLE: Definition of unsigned encoding

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$:

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i$$

$U_{min} = 00\ldots0_{w-1\ 0}$

$U_{max} = 11\ldots1_{w-1\ 0} = 2^w - 1$

PRINCIPLE: Definition of two's-complement encoding

For vector $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$:

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

T_{min} olması için bunun 0 olması lazım

↳ $10\ldots0_{w-1\ 0} = -2^{w-1}$

T_{max} olması için ilk terimin 0, ikincinin alabildiği en yüksek değer olması lazım

↳ $011\ldots1_{w-1\ 0} = 2^{w-1} - 1$

in Figures 2.9 and 2.10, the two's-complement range is asymmetric: $|TMin| = |TMax| + 1$; that is, there is no positive counterpart to $TMin$. As we shall see, this

source of subtle program bugs. This asymmetry arises because half the bit patterns (those with the sign bit set to 1) represent negative numbers, while half (those with the sign bit set to 0) represent nonnegative numbers. Since 0 is nonnegative, this means that it can represent one less positive number than negative. Second,

	Word size w			
Value	8	16	32	64
$UMax_w$	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
	255	65,535	4,294,967,295	18,446,744,073,709,551,615
$TMin_w$	0x80	0x8000	0x80000000	0x8000000000000000
	−128	−32,768	−2,147,483,648	−9,223,372,036,854,775,808
$TMax_w$	0x7F	0x7FFF	0x7FFFFFFF	0x7FFFFFFFFFFFFFFF
	127	32,767	2,147,483,647	9,223,372,036,854,775,807
−1	0xFF	0xFFFF	0xFFFFFFFF	0xFFFFFFFFFFFFFFFF
0	0x00	0x0000	0x00000000	0x0000000000000000

variable `x` is declared as `int` and `u` as `unsigned`. The expression `(unsigned) x` converts the value of `x` to an unsigned value, and `(int) u` converts the value of `u` to a signed integer. What should be the effect of casting signed value to unsigned,

zero. Converting an unsigned value that is too large to be represented in two's-complement form might yield $TMax$. For most implementations of C, however,

representation of 53,191. Casting from `short` to `unsigned short` changed the numeric value, but not the bit representation. →in C

PRINCIPLE: Conversion from two's complement to unsigned

For x such that $TMin_w \le x \le TMax_w$:

$$T2U_w(x) = \begin{cases} x + 2^w, & x < 0 \\ x, & x \ge 0 \end{cases} \tag{2.5}$$

For example, we saw that $T2U_{16}(-12{,}345) = -12{,}345 + 2^{16} = 53{,}191$, and also that $T2U_w(-1) = -1 + 2^w = UMax_w$.

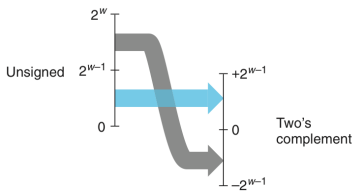
PRINCIPLE: Unsigned to two's-complement conversion

For u such that $0 \le u \le UMax_w$:

$$U2T_w(u) = \begin{cases} u, & u \le TMax_w \\ u - 2^w, & u > TMax_w \end{cases} \quad (2.7)$$

$2^{w-1} - 1 - 2^w$

Figure 2.18 Conversion from unsigned to two's complement. Function $U2T$ converts numbers greater than $2^{w-1} - 1$ to negative values.



For values outside of this range, the conversions either add or subtract 2^w. For example, we have $T2U_w(-1) = -1 + 2^w = UMax_w$—the negative number closest to zero maps to the largest unsigned number. At the other extreme, one can see that $T2U_w(TMin_w) = -2^{w-1} + 2^w = 2^{w-1} = TMax_w + 1$—the most negative number maps to an unsigned number just outside the range of positive two's-complement numbers. Using the example of Figure 2.15, we can see that

ity a particular representation of signed numbers, almost all machines use two's complement. Generally, most numbers are signed by default. For example, when declaring a constant such as `12345` or `0x1A2B`, the value is considered signed. Adding character 'U' or 'u' as a suffix creates an unsigned constant; for example, `12345U` or `0x1A2Bu`.

bits for the data type.

Conversions can happen due to explicit casting, such as in the following code:

```
1       int tx, ty;
2       unsigned ux, uy;
3
4       tx = (int) ux;
5       uy = (unsigned) ty;
```

Alternatively, they can happen implicitly when an expression of one type is assigned to a variable of another, as in the following code:

```
1       int tx, ty;
2       unsigned ux, uy;
3
4       tx = ux; /* Cast to signed */
5       uy = ty; /* Cast to unsigned */
```

When printing numeric values with `printf`, the directives `%d`, `%u`, and `%x` are used to print a number as a signed decimal, an unsigned decimal, and in hexadecimal format, respectively. Note that `printf` does not make use of any

Web Aside DATA:TMIN Writing $TMin$ in C

In Figure 2.19 and in Problem 2.21, we carefully wrote the value of $TMin_{32}$ as `-2,147,483,647-1`. Why not simply write it as either `-2,147,483,648` or `0x80000000`? Looking at the C header file `limits.h`, we see that they use a similar method as we have to write $TMin_{32}$ and $TMax_{32}$:

```
/* Minimum and maximum values a 'signed int' can hold.  */
#define INT_MAX   2147483647
#define INT_MIN   (-INT_MAX - 1)
```

Unfortunately, a curious interaction between the asymmetry of the two's-complement representation and the conversion rules of C forces us to write $TMin_{32}$ in this unusual way. Although understanding this issue requires us to delve into one of the murkier corners of the C language standards, it will help us appreciate some of the subtleties of integer data types and representations.

PRINCIPLE: Expansion of an unsigned number by zero extension

Define bit vectors $\vec{u} = [u_{w-1}, u_{w-2}, \ldots, u_0]$ of width w and $\vec{u}' = [0, \ldots, 0, u_{w-1}, u_{w-2}, \ldots, u_0]$ of width w', where $w' > w$. Then $B2U_w(\vec{u}) = B2U_{w'}(\vec{u}')$.

This principle can be seen to follow directly from the definition of the unsigned encoding, given by Equation 2.1.

For converting a two's-complement number to a larger data type, the rule is to perform a *sign extension*, adding copies of the most significant bit to the representation, expressed by the following principle. We show the sign bit x_{w-1} in blue to highlight its role in sign extension.

PRINCIPLE: Expansion of a two's-complement number by sign extension

Define bit vectors $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ of width w and $\vec{x}' = [x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]$ of width w', where $w' > w$. Then $B2T_w(\vec{x}) = B2T_{w'}(\vec{x}')$.

As an illustration, Figure 2.20 shows the result of expanding from word size $w = 3$ to $w = 4$ by sign extension. Bit vector [101] represents the value $-4 + 1 = -3$. Applying sign extension gives bit vector [1101] representing the value $-8 + 4 + 1 = -3$. We can see that, for $w = 4$, the combined value of the two most significant bits, $-8 + 4 = -4$, matches the value of the sign bit for $w = 3$. Similarly, bit vectors [111] and [1111] both represent the value -1.

```
short sx = -12345;       /* -12345   */
unsigned uy = sx;        /* Mystery! */

printf("uy  = %u:\t", uy);
show_bytes((byte_pointer) &uy, sizeof(unsigned));
```

change first size (sign extension / zero extension) then type (unsigned / T2U(x))

When run on a big-endian machine, this code causes the following output to be printed:

```
uy = 4294954951:  ff ff cf c7
```

This shows that, when converting from `short` to `unsigned`, the program first changes the size and then the type. That is, `(unsigned) sx` is equivalent to `(unsigned) (int) sx`, evaluating to 4,294,954,951, not `(unsigned) (unsigned short) sx`, which evaluates to 53,191. Indeed, this convention is required by the C standards.

İlk size'ı değiştirdiğim için sign extension yapmış oldum çünkü elimdeki sx hala short. İlk type değişmiş olsaydı zero extension yapmam gerekecekti.

When truncating a w-bit number $\vec{x} = [x_{w-1}, x_{w-2}, \ldots, x_0]$ to a k-bit number, we drop the high-order $w-k$ bits, giving a bit vector $\vec{x}' = [x_{k-1}, x_{k-2}, \ldots, x_0]$. Truncating a number can alter its value—a form of overflow. For an unsigned number, we can readily characterize the numeric value that will result.

PRINCIPLE: Truncation of an unsigned number Let's say from 32 bit to 16 bit

Let $\vec{x}$ be the bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and let $\vec{x}'$ be the result of truncating it to k bits: $\vec{x}' = [x_{k-1}, x_{k-2}, \ldots, x_0]$. Let $x = B2U_w(\vec{x})$ and $x' = B2U_k(\vec{x}')$. Then $x' = x \bmod 2^k$. ∎

↳ Yani bir binary sayının Least significant k tanesini bulmak istersen $x' = x \bmod 2^k$ (Least significant k bits)

PRINCIPLE: Unsigned addition

For x and y such that $0 \le x, y < 2^w$:

$$x +_w^u y = \begin{cases} x + y, & x + y < 2^w \quad \text{Normal} \\ x + y - 2^w, & 2^w \le x + y < 2^{w+1} \quad \text{Overflow} \end{cases} \tag{2.11}$$

↳ or $(x+y) \bmod 2^w$

PRINCIPLE: Detecting overflow of unsigned addition

For x and y in the range $0 \le x, y \le UMax_w$, let $s \doteq x +_w^u y$. Then the computation of s overflowed if and only if $s < x$ (or equivalently, $s < y$). ∎

DERIVATION: Detecting overflow of unsigned addition

Observe that $x + y \ge x$, and hence if s did not overflow, we will surely have $s \ge x$. On the other hand, if s did overflow, we have $s = x + y - 2^w$. Given that $y < 2^w$, we have $y - 2^w < 0$, and hence $s = x + (y - 2^w) < x$. ∎

PRINCIPLE: Unsigned negation

For any number x such that $0 \le x < 2^w$, its w-bit unsigned negation $-_w^u x$ is given by the following:

$$-_w^u x = \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases} \tag{2.12}$$

∎

DERIVATION: Unsigned negation

When $x = 0$, the additive inverse is clearly 0. For $x > 0$, consider the value $2^w - x$. Observe that this number is in the range $0 < 2^w - x < 2^w$. We can also see that $(x + 2^w - x) \bmod 2^w = 2^w \bmod 2^w = 0$. Hence it is the inverse of x under $+_w^u$. ∎

PRINCIPLE: Two's-complement addition

For integer values x and y in the range $-2^{w-1} \le x, y \le 2^{w-1} - 1$:

$$x +_w^t y = \begin{cases} x + y - 2^w, & 2^{w-1} \le x + y \quad \text{Positive overflow} \\ x + y, & -2^{w-1} \le x + y < 2^{w-1} \quad \text{Normal} \\ x + y + 2^w, & x + y < -2^{w-1} \quad \text{Negative overflow} \end{cases} \tag{2.13}$$

∎

Bu isimleri unsigned düşünebiliriz.
$[T2U_4(-8) + T2U_4(-5)] \bmod 2^4$
$= (8 + 11) \bmod 16$
$= 19 \bmod 16 = 3_{10} = 0011_2$

x	y	$x + y$	$x +_4^t y$	Case
−8 [1000]	−5 [1011]	(−13) [10011]	3 [0011]	1
−8 [1000]	−8 [1000]	−16 [10000]	0 [0000]	1
−8 [1000]	5 [0101]	−3 [11101]	−3 [1101]	2
2 [0010]	5 [0101]	7 [00111]	7 [0111]	3
5 [0101]	5 [0101]	10 [01010]	−6 [1010]	4

negative overflow ↳ $+2^w$ — $x+y < T_{min} \Rightarrow -13 < -2^3$

in this case, we reached to the same result because remember the sign extension. $10_2^3 = 1101_2^4 = 11101_2^5$

positive overflow ↳ -2^w — $x+y > T_{max} \Rightarrow 10 > 2^{4-1} - 1 = 2^3 - 1$, $10 > 7$, $\Rightarrow 10 - 2^4 = 10 - 16 = -6$

Figure 2.25 Two's-complement addition examples. The bit-level representation of the 4-bit two's-complement sum can be obtained by performing binary addition of the operands and truncating the result to 4 bits.

PRINCIPLE: Detecting overflow in two's-complement addition

For x and y in the range $TMin_w \leq x, y \leq TMax_w$, let $s \doteq x +^t_w y$. Then the computation of s has had positive overflow if and only if $x > 0$ and $y > 0$ but $s \leq 0$. The computation has had negative overflow if and only if $x < 0$ and $y < 0$ but $s \geq 0$.

Your coworker gets impatient with your analysis of the overflow conditions for two's-complement addition and presents you with the following implementation of `tadd_ok`:

```
/* Determine whether arguments can be added without overflow */
/* WARNING: This code is buggy. */
int tadd_ok(int x, int y) {
    int sum = x+y;
    return (sum-x == y) && (sum-y == x);
}
```

Your coworker could have learned, by studying Section 2.3.2, that two's-complement addition forms an abelian group, and so the expression `(x+y)-x` will evaluate to `y` regardless of whether or not the addition overflows, and that `(x+y)-y` will always evaluate to `x`.

You look at the code and laugh. Explain why.

We can see that every number x in the range $TMin_w \leq x \leq TMax_w$ has an additive inverse under $+^t_w$, which we denote $-^t_w x$ as follows:

PRINCIPLE: Two's-complement negation

For x in the range $TMin_w \leq x \leq TMax_w$, its two's-complement negation $-^t_w x$ is given by the formula

$$-^t_w x = \begin{cases} TMin_w, & x = TMin_w \\ -x, & x > TMin_w \end{cases} \tag{2.15}$$

DERIVATION: Two's-complement negation

Observe that $TMin_w + TMin_w = -2^{w-1} + -2^{w-1} = -2^w$. This would cause negative overflow, and hence $TMin_w +^t_w TMin_w = -2^w + 2^w = 0$. For values of x such that $x > TMin_w$, the value $-x$ can also be represented as a w-bit two's-complement number, and their sum will be $-x + x = 0$.

PRINCIPLE: Unsigned multiplication

For x and y such that $0 \leq x, y \leq UMax_w$:

$$x *^u_w y = (x \cdot y) \bmod 2^w \tag{2.16}$$

PRINCIPLE: Two's-complement multiplication

For x and y such that $TMin_w \leq x, y \leq TMax_w$:

$$x *^t_w y = U2T_w((x \cdot y) \bmod 2^w) \tag{2.17}$$

PRINCIPLE: Bit-level equivalence of unsigned and two's-complement multiplication

Let $\vec{x}$ and $\vec{y}$ be bit vectors of length w. Define integers x and y as the values represented by these bits in two's-complement form: $x = B2T_w(\vec{x})$ and $y = B2T_w(\vec{y})$. Define nonnegative integers x' and y' as the values represented by these bits in unsigned form: $x' = B2U_w(\vec{x})$ and $y' = B2U_w(\vec{y})$. Then

$$T2B_w(x *^t_w y) = U2B_w(x' *^u_w y')$$

to 3 bits. The unsigned truncated product always equals $x \cdot y \bmod 8$. The bit-level representations of both truncated products are identical for both unsigned and two's-complement multiplication, even though the full 6-bit representations differ.

Mode	x		y		$x \cdot y$		Truncated $x \cdot y$	
Unsigned	5	[101]	3	[011]	15	[001111]	7	[111]
Two's complement	−3	[101]	3	[011]	−9	[110111]	−1	[111]
Unsigned	4	[100]	7	[111]	28	[011100]	4	[100]
Two's complement	−4	[100]	−1	[111]	4	[000100]	−4	[100]
Unsigned	3	[011]	3	[011]	9	[001001]	1	[001]
Two's complement	3	[011]	3	[011]	9	[001001]	1	[001]

PRINCIPLE: Multiplication by a power of 2

Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$. Then for any $k \geq 0$, the $w + k$-bit unsigned representation of $x2^k$ is given by $[x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]$, where k zeros have been added to the right.

PRINCIPLE: Unsigned multiplication by a power of 2

For C variables x and k with unsigned values x and k, such that $0 \leq k < w$, the C expression x << k yields the value $x *_w^u 2^k$.

PRINCIPLE: Two's-complement multiplication by a power of 2

For C variables x and k with two's-complement value x and unsigned value k, such that $0 \leq k < w$, the C expression x << k yields the value $x *_w^t 2^k$.

C compilers try to remove many cases where an integer is being multiplied by a constant with combinations of shifting, adding, and subtracting. For example, suppose a program contains the expression x*14. Recognizing that $14 = 2^3 + 2^2 + 2^1$, the compiler can rewrite the multiplication as `(x<<3) + (x<<2) + (x<<1)`, replacing one multiplication with three shifts and two additions. The two computations will yield the same result, regardless of whether x is unsigned or two's complement, and even if the multiplication would cause an overflow. Even better, the compiler can also use the property $14 = 2^4 - 2^1$ to rewrite the multiplication as `(x<<4) - (x<<1)`, requiring only two shifts and a subtraction.

Generalizing from our example, consider the task of generating code for the expression x * K, for some constant K. The compiler can express the binary representation of K as an alternating sequence of zeros and ones:

$$[(0\ldots0)\ (1\ldots1)\ (0\ldots0)\cdots(1\ldots1)]$$

For example, 14 can be written as $[(0\ldots0)(111)(0)]$. Consider a run of ones from bit position n down to bit position m ($n \geq m$). (For the case of 14, we have $n = 3$ and $m = 1$.) We can compute the effect of these bits on the product using either of two different forms:

Form A: `(x<<`n`) + (x<<(`$n-1$`)) + ··· + (x<<`m`)`

Form B: `(x<<(`$n+1$`)) - (x<<`m`)`

k	>> k (binary)	Decimal	$12{,}340/2^k$
0	0011000000110100	12,340	12,340.0
1	0001100000011010	6,170	6,170.0
4	0000001100000011	771	771.25
8	0000000000110000	48	48.203125

Figure 2.28 Dividing unsigned numbers by powers of 2. The examples illustrate how performing a logical right shift by k has the same effect as dividing by 2^k and then rounding toward zero.

Integer division always rounds toward zero. To define this precisely, let us

As examples, $\lceil 3.14 \rceil = 4$, $\lceil -3.14 \rceil = -3$, and $\lceil 3 \rceil = 3$. For $x \geq 0$ and $y > 0$, integer division should yield $\lfloor x/y \rfloor$, while for $x < 0$ and $y > 0$, it should yield $\lceil x/y \rceil$. That is, it should round down a positive result but round up a negative one.

PRINCIPLE: Unsigned division by a power of 2

For C variables x and k with unsigned values x and k, such that $0 \leq k < w$, the C expression x >> k yields the value $\lfloor x/2^k \rfloor$.

k	>> k (binary)	Decimal	$-12{,}340/2^k$
0	1100111111001100	−12,340	−12,340.0
1	1111001111100110	−6,170	−6,170.0
4	1111110011111100	−772	−771.25
8	1111111111001111	−49	−48.203125

Figure 2.29 Applying arithmetic right shift. The examples illustrate that arithmetic right shift is similar to division by a power of 2, except that it rounds down rather than toward zero.

k	Bias	−12,340 + bias (binary)	>> k (binary)	Decimal	$-12{,}340/2^k$
0	0	1100111111001100	1100111111001100	−12,340	−12,340.0
1	1	1100111111001101	1110011111100110	−6,170	−6,170.0
4	15	1100111111011011	1111110011111101	−771	−771.25
8	255	1101000011001011	1111111111010000	−48	−48.203125

Figure 2.30 Dividing two's-complement numbers by powers of 2. By adding a bias before the right shift, the result is rounded toward zero.

PRINCIPLE: Two's-complement division by a power of 2, rounding up

Let C variables x and k have two's-complement value x and unsigned value k, respectively, such that $0 \leq k < w$. The C expression `(x + (1 << k) - 1) >> k`, when the shift is performed arithmetically, yields the value $\lceil x/2^k \rceil$.

These analyses show that for a two's-complement machine using arithmetic right shifts, the C expression

```
(x<0 ? x+(1<<k)-1 : x) >> k
```

will compute the value $x/2^k$.

Floating Point

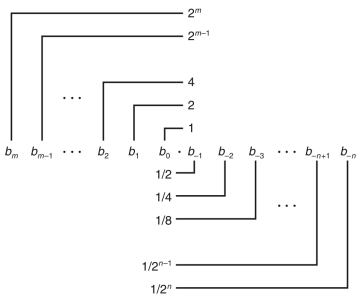


Figure 2.31 Fractional binary representation. Digits to the left of the binary point have weights of the form 2^i, while those to the right have weights of the form $1/2^i$.

By analogy, consider a notation of the form

$$b_m\, b_{m-1} \cdots b_1\, b_0 \,.\, b_{-1}\, b_{-2} \cdots b_{-n+1}\, b_{-n}$$

where each binary digit, or bit, b_i ranges between 0 and 1, as is illustrated in Figure 2.31. This notation represents a number b defined as

$$b = \sum_{i=-n}^{m} 2^i \times b_i \tag{2.19}$$

The symbol '.' now becomes a *binary point*, with bits on the left being weighted by nonnegative powers of 2, and those on the right being weighted by negative powers of 2. For example, 101.11_2 represents the number $1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} = 4 + 0 + 1 + \frac{1}{2} + \frac{1}{4} = 5\frac{3}{4}$.

One can readily see from Equation 2.19 that shifting the binary point one position to the left has the effect of dividing the number by 2. For example, while $5 + 0 + 2 + 1 + \frac{1}{2} = 11\frac{1}{2}$.

Note that numbers of the form $0.11\cdots1_2$ represent numbers just below 1. For example, 0.111111_2 represents $\frac{63}{64}$. We will use the shorthand notation $1.0 - \epsilon$ to represent such values.

One simple way to think about fractional binary representations is to represent a number as a fraction of the form $\frac{x}{2^k}$. We can write this in binary using the binary representation of x, with the binary point inserted k positions from the right. As an example, for $\frac{25}{16}$, we have $25_{10} = 11001_2$. We then put the binary point four positions from the right to get 1.1001_2.

The IEEE floating-point standard represents a number in a form $V = (-1)^s \times M \times 2^E$:

- The *sign* s determines whether the number is negative ($s = 1$) or positive ($s = 0$), where the interpretation of the sign bit for numeric value 0 is handled as a special case.
- The *significand* M is a fractional binary number that ranges either between 1 and $2 - \epsilon$ or between 0 and $1 - \epsilon$.
- The *exponent* E weights the value by a (possibly negative) power of 2.

The bit representation of a floating-point number is divided into three fields to encode these values:

- The single sign bit `s` directly encodes the sign s.
- The k-bit exponent field `exp` $= e_{k-1} \cdots e_1 e_0$ encodes the exponent E.
- The n-bit fraction field `frac` $= f_{n-1} \cdots f_1 f_0$ encodes the significand M, but the value encoded also depends on whether or not the exponent field equals 0.

Figure 2.32 shows the packing of these three fields into words for the two most common formats. In the single-precision floating-point format (a `float` in C), fields `s`, `exp`, and `frac` are 1, $k = 8$, and $n = 23$ bits each, yielding a 32-bit representation. In the double-precision floating-point format (a `double` in C), fields `s`, `exp`, and `frac` are 1, $k = 11$, and $n = 52$ bits each, yielding a 64-bit representation.

Case 1: Normalized Values

This is the most common case. It occurs when the bit pattern of `exp` is neither all zeros (numeric value 0) nor all ones (numeric value 255 for single precision, 2047 for double). In this case, the exponent field is interpreted as representing a signed integer in *biased* form. That is, the exponent value is $E = e - Bias$, where e is the unsigned number having bit representation $e_{k-1} \cdots e_1 e_0$ and *Bias* is a bias value equal to $2^{k-1} - 1$ (127 for single precision and 1023 for double). This yields exponent ranges from -126 to $+127$ for single precision and -1022 to $+1023$ for double precision.

The fraction field `frac` is interpreted as representing the fractional value f, where $0 \le f < 1$, having binary representation $0.f_{n-1} \cdots f_1 f_0$, that is, with the

binary point to the left of the most significant bit. The significand is defined to be $M = 1 + f$. This is sometimes called an *implied leading 1* representation, because we can view M to be the number with binary representation $1.f_{n-1}f_{n-2}\cdots f_0$. This representation is a trick for getting an additional bit of precision for free, since we can always adjust the exponent E so that significand M is in the range $1 \le M < 2$ (assuming there is no overflow). We therefore do not need to explicitly represent the leading bit, since it always equals 1.

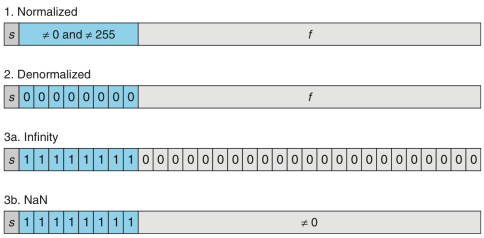


Figure 2.33 **Categories of single-precision floating-point values.** The value of the exponent determines whether the number is (1) normalized, (2) denormalized, or (3) a special value.

Case 2: Denormalized Values

When the exponent field is all zeros, the represented number is in *denormalized* form. In this case, the exponent value is $E = 1 - Bias$, and the significand value is $M = f$, that is, the value of the fraction field without an implied leading 1.