Operation	Result
a	$[01101001]$
b	$[01010101]$
~a	
~b	
a & b	
a | b	
a ^ b	

One useful application of bit vectors is to represent finite sets. For example, we can denote any subset $A \subseteq \{0, 1, \ldots, w-1\}$ as a bit vector $[a_{w-1}, \ldots, a_1, a_0]$, where $a_i = 1$ if and only if $i \in A$. For example, (recalling that we write a_{w-1} on the left and a_0 on the right), we have $a = [01101001]$ representing the set $A = \{0, 3, 5, 6\}$, and $b = [01010101]$ representing the set $B = \{0, 2, 4, 6\}$. Under this interpretation, Boolean operations | and & correspond to set union and intersection, respectively, and ~ corresponds to set complement. For example, the operation a & b yields bit vector $[01000001]$, while $A \cap B = \{0, 6\}$.

In fact, for any set S, the structure $\langle \mathcal{P}(S), \cup, \cap, \overline{\ }, \emptyset, S \rangle$ forms a Boolean algebra, where $\mathcal{P}(S)$ denotes the set of all subsets of S, and $\overline{\ }$ denotes the set complement operator. That is, for any set A, its complement is the set $\overline{A} = \{a \in S | a \notin A\}$. The ability to represent and manipulate finite sets using bit vector operations is a practical outcome of a deep mathematical principle.

2.1.8 Bit-Level Operations in C

One useful feature of C is that it supports bit-wise Boolean operations. In fact, the symbols we have used for the Boolean operations are exactly those used by C: | for OR, & for AND, ~ for NOT, and ^ for EXCLUSIVE-OR. These can be applied to any "integral" data type, that is, one declared as type `char` or `int`, with or without qualifiers such as `short`, `long`, or `unsigned`. Here are some example expression evaluations:

C Expression	Binary Expression	Binary Result	C Result
`~0x41`	~$[01000001]$	$[10111110]$	`0xBE`
`~0x00`	~$[00000000]$	$[11111111]$	`0xFF`
`0x69 & 0x55`	$[01101001]$ & $[01010101]$	$[01000001]$	`0x41`
`0x69 | 0x55`	$[01101001]$ | $[01010101]$	$[01111101]$	`0x7D`

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.

Practice Problem 2.6:

To show how the ring properties of ^ can be useful, consider the following program:

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

```
4      *y = *x ^ *y;  /* Step 2 */
5      *x = *x ^ *y;  /* Step 3 */
6  }
```

As the name implies, we claim that the effect of this procedure is to swap the values stored at the locations denoted by pointer variables `x` and `y`. Note that unlike the usual technique for swapping two values, we do not need a third location to temporarily store one value while we are moving the other. There is no performance advantage to this way of swapping. It is merely an intellectual amusement.

Starting with values a and b in the locations pointed to by `x` and `y`, respectively, fill in the following table giving the values stored at the two locations after each step of the procedure. Use the ring properties to show that the desired effect is achieved. Recall that every element is its own additive inverse, that is, $a \hat{} a = 0$.

Step	`*x`	`*y`
Initially	a	b
Step 1		
Step 2		
Step 3		

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 1s for the least significant eight 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 1s, regardless of the word size of the machine. Although the same mask can be written `0xFFFFFFFF` for a 32-bit machine, such code is not as portable.

Practice Problem 2.7:

Write C expressions for the following values, with the results for `x` = `0x98FDECBA` and a 32-bit word size shown in square brackets:

A. The least significant byte of `x`, with all other bits set to 1 [`0xFFFFFFBA`].

B. The complement of the least significant byte of `x`, with all other bytes left unchanged [`0x98FDEC45`].

C. All but the least significant byte of `x`, with the least significant byte set to 0 [`0x98FDEC00`].

Although our examples assume a 32-bit word size, your code should work for any word size $w \geq 8$.

Practice Problem 2.8:

The Digital Equipment VAX computer was a very popular machine from the late 1970s until the late 1980s. Rather than instructions for Boolean operations AND and OR, it had instructions `bis` (bit set) and `bic` (bit clear). Both instructions take a data word `x` and a mask word `m`. They generate a result `z` consisting of the bits of `x` modified according to the bits of `m`. With `bis`, the modification involves setting `z` to 1 at each bit position where `m` is 1. With `bic`, the modification involves setting `z` to 0 at each bit position where `m` is 1.

We would like to write C functions `bis` and `bic` to compute the effect of these two instructions. Fill in the missing expressions in the code below using the bit-level operations of C.

```
/* Bit Set */
int bis(int x, int m)
{
  /* Write an expression in C that computes the effect of bit set */
  int result = ____________;
  return result;
}

/* Bit Clear */
int bic(int x, int m)
{
  /* Write an expression in C that computes the effect of bit set */
  int result = ____________;
  return result;
}
```

2.1.9 Logical Operations in C

C also provides a set of *logical* operators `||`, `&&`, and `!`, which correspond to the OR, AND, and NOT operations of propositional logic. These can easily be confused with the bit-level operations, but their function is quite different. The logical operations treat any nonzero argument as representing TRUE and argument 0 as representing FALSE. They return either 1 or 0 indicating a result of either TRUE or FALSE, respectively. Here are some example expression evaluations:

Expression	Result
`!0x41`	`0x00`
`!0x00`	`0x01`
`!!0x41`	`0x01`
`0x69 && 0x55`	`0x01`
`0x69 || 0x55`	`0x01`

Observe that a bit-wise operation will have behavior matching that of its logical counterpart only in the special case where the arguments are restricted to be either 0 or 1.

A second important distinction between the logical operators `&&` and `||`, versus their bit-level counterparts `&` and `|` is that the logical operators do not evaluate their second argument if the result of the expression can be determined by evaluating the first argument. Thus, for example, the expression `a && 5/a` will never cause a division by zero, and the expression `p && *p++` will never cause the dereferencing of a null pointer.

Practice Problem 2.9:

Suppose that `x` and `y` have byte values `0x66` and `0x93`, respectively. Fill in the following table indicating the byte values of the different C expressions

Expression	Value	Expression	Value
`x & y`		`x && y`	
`x | y`		`x || y`	
`~x | ~y`		`!x || !y`	
`x & !y`		`x && ~y`	

Practice Problem 2.10:

Using only bit-level and logical operations, write a C expression that is equivalent to `x == y`. That is, it will return 1 when `x` and `y` are equal and 0 otherwise.

2.1.10 Shift Operations in C

C also provides a set of *shift* operations for shifting bit patterns to the left and to the right. For an operand `x` having bit representation $[x_{n-1}, x_{n-2}, \ldots, x_0]$, the C expression `x << k` yields a value with bit representation $[x_{n-k-1}, x_{n-k-2}, \ldots, x_0, 0, \ldots 0]$. That is, `x` is shifted k bits to the left, dropping off the k most significant bits and filling the left end with k 0s. The shift amount should be a value between 0 and $n-1$. Shift operations group from left to right, so `x << j << k` is equivalent to `(x << j) << k`. Be careful about operator precedence: `1<<5 - 1` is evaluated as `1 << (5-1)`, not as `(1<<5) - 1`.

There is a corresponding right shift operation `x >> k`, but it has a slightly subtle behavior. Generally, machines support two forms of right shift: *logical* and *arithmetic*. A logical right shift fills the left end with k 0s, giving a result $[0, \ldots, 0, x_{n-1}, x_{n-2}, \ldots x_k]$. An arithmetic right shift fills the left end with k repetitions of the most significant bit, giving a result $[x_{n-1}, \ldots, x_{n-1}, x_{n-1}, x_{n-2}, \ldots x_k]$. This convention might seem peculiar, but as we will see it is useful for operating on signed integer data.

The C standard does not precisely define which type of right shift should be used. For unsigned data (i.e., integral objects declared with the qualifier `unsigned`), right shifts must be logical. For signed data (the default), either arithmetic or logical shifts may be used. This unfortunately means that any code assuming one form or the other will potentially encounter portability problems. In practice, however, almost all compiler/machine combinations use arithmetic right shifts for signed data, and many programmers assume this to be the case.

Practice Problem 2.11:

Fill in the table below showing the effects of the different shift operations on single-byte quantities. Write each answer as two hexadecimal digits.

`x`	`x << 3`	`x >> 2` (Logical)	`x >> 2` (Arithmetic)
`0xF0`			
`0x0F`			
`0xCC`			
`0x55`			

C Declaration	Guaranteed		Typical 32-bit	
	Minimum	Maximum	Minimum	Maximum
`char`	−127	127	−128	127
`unsigned char`	0	255	0	255
`short [int]`	−32,767	32,767	−32,768	32,767
`unsigned short [int]`	0	63,535	0	63,535
`int`	−32,767	32,767	−2,147,483,648	2,147,483,647
`unsigned [int]`	0	65,535	0	4,294,967,295
`long [int]`	−2,147,483,647	2,147,483,647	−2,147,483,648	2,147,483,647
`unsigned long [int]`	0	4,294,967,295	0	4,294,967,295

Figure 2.8: **C Integral Data types.** Text in square brackets is optional.

2.2 Integer Representations

In this section we describe two different ways bits can be used to encode integers—one that can only represent nonnegative numbers, and one that can represent negative, zero, and positive numbers. We will see later that they are strongly related both in their mathematical properties and their machine-level implementations. We also investigate the effect of expanding or shrinking an encoded integer to fit a representation with a different length.

2.2.1 Integral Data Types

C supports a variety of *integral* data types—ones that represent a finite range of integers. These are shown in Figure 2.8. Each type has a size designator: `char`, `short`, `int`, and `long`, as well as an indication of whether the represented number is nonnegative (declared as `unsigned`), or possibly negative (the default). The typical allocations for these different sizes were given in Figure 2.2. As indicated in Figure 2.8, these different sizes allow different ranges of values to be represented. The C standard defines a minimum range of values each data type must be able to represent. As shown in the figure, a typical 32-bit machine uses a 32-bit representation for data types `int` and `unsigned`, even though the C standard allows 16-bit representations. As described in Figure 2.2, the Compaq Alpha uses a 64-bit word to represent `long` integers, giving an upper limit of over 1.84×10^{19} for unsigned values, and a range of over $\pm 9.22 \times 10^{18}$ for signed values.

New to C?
Both C and C++ support signed (the default) and unsigned numbers. Java supports only signed numbers. **End**

2.2.2 Unsigned and Two's Complement Encodings

Assume we have an integer data type of w bits. We write a bit vector as either $\vec{x}$, to denote the entire vector, or as $[x_{w-1}, x_{w-2}, \ldots, x_0]$ to denote the individual bits within the vector. Treating $\vec{x}$ as a number written in binary notation, we obtain the *unsigned* interpretation of $\vec{x}$. We express this interpretation as a function