Chapter 2

Representing and Manipulating Information

2.1 Information Storage

Modern computers store and process information represented as two-valued signals. These lowly binary digits, or *bits* form the basis of the digital revolution. It may seem wasteful to transmit a two-valued signal along a wire, as opposed to say using 10 different voltage levels to encode a decimal digit. The circuitry to transmit and store two-valued signals is much simpler and more reliable, enabling manufacturers to integrate millions of such circuits on a single silicon chip.

Rather than accessing individual bits in a memory, most computers use blocks of 8 bits, or *bytes* as the smallest addressable unit of memory. That is, a machine-level program views its memory space as a very large array of bytes. As indicated by its name, this *virtual address space* is merely a conceptual image presented to the machine-level program. The actual implementation uses a combination of random-access memory (RAM), disk storage, special hardware and operating system software to implement a large, "flat" address space.

One task of a compiler and a program's run-time system is to subdivide this memory space into more manageable units to store individual program data, instructions, and control information. Various mechanisms are used to allocate and manage the storage for different parts of the program. This management is all performed within the virtual address space. For example, a pointer in C, whether it points to an integer, a structure, or some other program unit, is simply the virtual address of the first byte of some block of storage.

2.1.1 Hexadecimal Notation

A single byte consists of 8 bits. In binary notation, its value ranges from 00000000_2 to 11111111_2. When viewed as a decimal integer, its value ranges from 0_{10} to 255_{10}. Neither notation is very convenient for describing bit patterns. Binary notation is too verbose, while with decimal notation, it is tedious to convert to and from bit patterns. Instead, we write bit patterns as base 16, or *hexadecimal* numbers. Hexadecimal (or simply "Hex") uses digits '0' through '9', along with characters 'A' through 'F' to represent 16 possible

Hex digit	0	1	2	3	4	5	6	7
Decimal Value	0	1	2	3	4	5	6	7
Binary Value	0000	0001	0010	0011	0100	0101	0110	0111

Hex digit	8	9	A	B	C	D	E	F
Decimal Value	8	9	10	11	12	13	14	15
Binary Value	1000	1001	1010	1011	1100	1101	1110	1111

Table 2.1: **Hexadecimal Notation** Each Hex digit encodes one of 16 values.

values. Table 2.1 shows the decimal and binary values associated with the 16 hexadecimal digits. Written in hexadecimal, the value of a single byte can range from 00_{16} to FF_{16}.

In C, numeric constants starting with `0x` or `0X` are interpreted as being in hexadecimal. The characters 'A' through 'F' may be written in either upper or lower case. For example, we could write the number $FA1D37B_{16}$ as `0xFA1D37B`, as `0xfa1d37b`, or even mixing upper and lower case, e.g., `0xFa1D37b`. From now on, we will use the C notation for representing hexadecimal values in this text.

An unpleasant, but often necessary task in working with machine-level programs is to manually convert between decimal, binary, and hexadecimal representations of bit patterns. A starting point is to be able to convert, in both directions, between a single hexadecimal digit and a 4-bit binary pattern.

Problem 2.1 [Category 1]:
Fill in the missing entries in the following table, giving the decimal, binary, and hexadecimal values of different byte patterns.

Decimal	Binary	Hexadecimal
0	00000000	00
55		
136		
243		
	01010010	
	10101100	
	11100111	
		A7
		3E
		BC

2.1.2 Words

Every computer has a *word size*, indicating the nominal size of integer-valued data. Since a virtual address is encoded by such a word, the most important system parameter determined by the word size is the size of the virtual address space. That is, for a machine with an n-bit word size, the virtual addresses can range

C Declaration	Typical 32-bit	Compaq Alpha
`char`	1	1
`int`	4	4
`short int`	2	2
`long int`	4	8
`char *`	4	8
`float`	4	4
`double`	8	8

Table 2.2: **Sizes (in Bytes) of C Numeric Data Types.** The number of bytes allocated varies with machine and compiler.

from 0 to $2^n - 1$, giving the program access to at most 2^n bytes.

Most computers today have a 32-bit word size. This limits the virtual address space to 4 gigabytes (4GB), i.e., just over 4×10^9 bytes. Although this is ample space for most applications, we are approaching the point where many large scale scientific and database applications require larger amounts of storage. Consequently, high-end machines with 64-bit word sizes are becoming available. Such machines will become increasingly commonplace as storage costs decrease.

2.1.3 Data Sizes

Computers and compilers support multiple data formats using different encodings (e.g., integers and floating point) and different lengths. For example, many machines have instructions for manipulating single bytes, as well as integers represented as 2, 4, and 8-byte quantities. They also support floating point numbers represented as 4 and 8-byte quantities.

The C language also supports multiple data formats for both integer and floating point data. The C data type `char` represents a single byte. Although the name "char" derives from the fact that it is used to store a single character in a text string, it can also be used to store integer values. The C data type `int` can also be prefixed by the qualifiers `long` and `short`, providing integer representations of various sizes. Table 2.2 shows the number of bytes allocated for various C data types. The exact number can depend on both the machine and the compiler. We show two representative cases: a typical 32-bit machine, and the Compaq Alpha architecture, a 64-bit machine targeting high end applications. Most 32-bit machines use the allocations indicated as "typical." Observe that "short" integers have 2-byte allocations, while an unqualified `int` is 4 bytes. A "long" integer uses the full word size of the machine.

Table 2.2 also shows that a pointer (e.g., a variable declared as being of type "`char *`") also uses the full word size of the machine. In addition, most machines support two different floating point formats: single precision, declared in C as `float`, and double precision, declared in C as `double`. These formats use 4 and 8 bytes, respectively.

Programmers should strive, as much as possible, to make their programs portable across different machines and compilers. One aspect of portability is to make the program insensitive to the exact sizes of the different data types. The C standard sets lower bounds on the numeric ranges of the different data types, as will be covered later, but there are no upper upper bounds. Since 32-bit machines have been the standard for the last 20 years, many programs have been written assuming the allocations listed as "typical 32-bit" in Table 2.2. Given the increasing prominence of 64-bit machines in the near future, many hidden word size dependencies will show up as bugs in migrating these programs to new machines. For example, many programmers assume that a program object declared as type `int` can be used to store a pointer. This works fine for most 32-bit machines, but leads to problems on an Alpha.

2.1.4 Addressing and Byte Ordering

For program objects that span multiple bytes, we must establish two conventions: what will be the address of the object, and how will we order the bytes in memory. In virtually all machines, a multi-byte object is stored as a contiguous sequence of bytes, with the address of the object given by the smallest address of the bytes used. For example, suppose a variable `x` of type `int` has address `0x100`, i.e., the value of the address expression `&x` is `0x100`. Then the 4 bytes of `x` would be stored in memory locations `0x100`, `0x101`, `0x102`, and `0x103`.

For ordering the bytes representing an object, there are two common conventions. Consider a w-bit integer having a bit representation $[x_{w-1}, x_{w-2}, \ldots, x_1, x_0]$, where x_{w-1} is the most significant bit, and x_0 is the least. These bits are then grouped as bytes, with the most significant byte having bits $[x_{w-1}, x_{w-2}, \ldots, x_{w-8}]$, the least significant byte having bits $[x_7, x_6, \ldots, x_0]$, and the other bytes having bits from the middle. Some machines choose to store the object in memory ordered from most significant byte to least, while other machines store them from least to most. The former convention, where the least significant byte comes first, is referred to as *Little Endian*. This convention is followed by most machines from the former Digital Equipment Corporation, as well as from Intel. The latter convention, where the most significant byte comes first, is referred to as *Big Endian*. This convention is followed by most machines from IBM, Motorola, and Sun Microsystems. Note that the above classification of companies uses the qualifier "most," since the conventions do not split precisely along corporate boundaries. For example, personal computers manufactured by IBM use the Intel chip and hence are little endian. Many microprocessor chips, including Alpha and the PowerPC by Motorola can be run in either mode, with the byte ordering convention determined when the chip is powered up.

Continuing our earlier example, suppose the variable `x` of type `int` and address `0x100` has hexadecimal value is `0x01234567`, and the value of address expression `&x` is `0x100`. The ordering of the bytes within the address range `0x100` through `0x103` would depend on the type of machine:

Big Endian

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

Little Endian

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

Note that in the word `0x01234567` the high order byte has hexadecimal value `0x01`, while the low order byte has value `0x67`.

```
typedef unsigned char *byte_pointer;

void show_bytes(byte_pointer start, int len)
{
  int i;
  for (i = 0; i < len; i++)
    printf(" %.2x", start[i]);
  printf("\n");
}

void show_int(int x)
{
  show_bytes((byte_pointer) &x, sizeof(int));
}

void show_float(float x)
{
  show_bytes((byte_pointer) &x, sizeof(float));
}

void show_pointer(void *x)
{
  show_bytes((byte_pointer) &x, sizeof(void *));
}
```

Figure 2.1: **Code to Print the Byte Representation of Program Objects.** This code uses casting to circumvent the type system.

People get surprisingly emotional about which byte ordering is the proper one. In fact, the names "Little Endian" and "Big Endian" come from the book *Gulliver's Travels* by Jonathan Swift, where two warring factions couldn't agree by which end a soft-boiled egg should be opened—the little end or the big. Just like the egg issue, there is no technological reason to choose one byte ordering convention over the other, and hence the arguments degenerate into bickering about socio-political issues.

For most application programmers, the byte ordering used by their machine is totally transparent. Programs can be compiled for either class of machine and give identical results. There are cases, however, in which byte ordering becomes an issue. The first is when binary data is communicated over a network between different machines. A common problem is for data produced by a little endian machine to be sent to a big endian machine, or vice-versa, leading to the bytes within the words being in reverse order for the receiving program. To avoid such problems, code written for networking applications should establish conventions for byte ordering and make sure the sending machine converts its internal representation to the network standard, while the receiving machine converts the network standard to its internal representation.

A second case is when programs are written that circumvent the normal type system. In the C language, this

```
int ival = 12345;
float fval = 12345;
int *pval = &ival;
show_int(ival);
show_float(fval);
show_pointer(pval);
```

Figure 2.2: **Byte Representation Examples.** This code prints the byte representations of sample data objects.

can be done using a *cast* to allow an object to be referenced according to a different data type from which it was created. Such coding tricks are strongly discouraged for most application programming, but they can be quite useful and even necessary for system-level programming.

Figure 2.1 shows C code that uses casting to access and print the byte representations of different program objects. We use `typedef` to define data type `byte_pointer` as pointer to an object of type "`unsigned char`." Such a byte pointer references a sequence of bytes where each byte is considered to be a nonnegative integer. The first routine `show_bytes` is given the address of a sequence of bytes, indicated by a byte pointer, and a byte count. It prints the individual bytes in hexadecimal. The C formatting directive "`%.2x`" indicates that an integer should be printed in hexadecimal with at least two digits.

Procedures `show_int`, `show_float`, and `show_pointer` demonstrate how to use procedure `show_bytes` to print the byte representations of C program objects of type `int`, `float`, and `void *`, respectively. Observe that they simply pass `show_bytes` a pointer `&x` to their argument `x`, casting the pointer to be of type "`unsigned char *`." This cast indicates to the compiler that the program should consider the pointer to be to a sequence of bytes rather than to an object of the original data type. This pointer will then be to the lowest byte address used by the object.

These procedures use the C operator `sizeof` to determine the number of bytes used by the object. In general, the expression `sizeof(`T`)` returns the number of bytes required to store an object of type T. Using `sizeof`, rather than a fixed value, is one step toward writing code that is portable across different machine types.

We ran the code shown in Figure 2.2 on several different machines, giving the results shown in Figure 2.3. The machines used were:

Linux: Intel Pentium II running Linux.

NT: Intel Pentium II running Windows-NT.

Sun: Sun Microsystems UltraSPARC running Solaris.

Alpha: Compaq Alpha 21164 running Digital Unix.

Our sample integer argument 12345 has hexadecimal representation `0x00003039`. For the `int` data, we get identical results for all machines, except for the byte ordering. In particular, we can see that the least significant byte value of `0x39` is printed first for Linux, NT, and Alpha, indicating little endian machines,

Machine	Value	Type	Bytes (Hex)
Linux	12345	`int`	`39 30 00 00`
NT	12345	`int`	`39 30 00 00`
Sun	12345	`int`	`00 00 30 39`
Alpha	12345	`int`	`39 30 00 00`
Linux	12345.	`float`	`00 e4 40 46`
NT	12345.	`float`	`00 e4 40 46`
Sun	12345.	`float`	`46 40 e4 00`
Alpha	12345.	`float`	`00 e4 40 46`
Linux	`&ival`	`int *`	`3c fa ff bf`
NT	`&ival`	`int *`	`1c ff 44 02`
Sun	`&ival`	`int *`	`ef ff fc e4`
Alpha	`&ival`	`int *`	`80 fc ff 1f 01 00 00 00`

Figure 2.3: **Byte Representations of Different Data Values.** Results for `int` and `float` are identical, except for byte ordering. Pointer values are machine-dependent.

and last for Sun, indicating a big endian machine. Similarly, the bytes of the `float` data are identical, except for the byte ordering. On the other hand, the pointer values are completely different. The different machine/operating system configurations use different conventions for storage allocation. One interesting feature to note is that the Linux and Sun machines use 4-byte addresses, while the Alpha uses 8-byte addresses.

Observe that although the floating point and the integer data both represent the numeric value 12345, they have very different byte patterns: `0x00007AB7` for the integer, and `0x4640E4` for floating point. In general, these two formats use different encoding schemes. Observe, however, that if we expand these hexadecimal patterns into binary and shift them appropriately, we find a sequence of 13 matching bits, indicated by a sequence of asterisks:

```
   0    0    0    0    3    0    3    9
00000000000000000011000000111001
                     *************
            4    6    4    0    E    4    0    0
         01000110010000001110010000000000
```

This is not coincidental. We shall return to this example when we study floating point formats.

Problem 2.2 [Category 1]:
Run the sample code on different machines to which you have access. Determine the byte orderings used by these machines.

Problem 2.3 [Category 1]:
Try running the code for different sample values.

Problem 2.4 [Category 1]:
Write procedures `show_short`, `show_long`, and `show_double` that print the byte representations of C objects of types `short int`, `long int`, and `double` respectively. Try these out on several machines.

Problem 2.5 [Category 2]:
Write a procedure `is_little_endian` that will return 1 when compiled and run on a little endian machine, and will return 0 when compiled and run on a big endian machine.

2.1.5 Representing Strings

Strings in C are denoted by arrays of characters terminated by the null character. Each character is represented by some standard encoding, with the most common being the ASCII character code. Thus, if we run our routine `show_bytes` with arguments `"12345"` and `6` (to include the terminating character), we get the result `33 31 34 31 35 00`. Observe that the ASCII code for digit value x happens to be `0x3`x, and that the terminating byte has hex representation `0x00`. This same result would be obtained on any system using ASCII as its character code, independent of the byte ordering and word size conventions. As a consequence, text data is more platform-independent than binary data.

2.1.6 Representing Code

Consider the following C function:

```
int sum(int x, int y)
{
  return x+y;
}
```

When compiled on our sample machines, we generate machine code having the following byte representations:

Linux: `55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3`

NT: `55 89 e5 8b 45 0c 03 45 08 89 ec 5d c3`

Sun: `81 C3 E0 08 90 02 00 09`

Alpha: `00 00 30 42 01 80 FA 6B`

Here we find that the instruction codings are different, except for the NT and Linux machines. In general, different machine types use different and incompatible instructions and encodings. The NT and Linux machines both have Intel processors and hence support the same machine-level instructions. In general, however, the structure of an executable NT program differs from a Linux program, and hence the machines

~	
0	1
1	0

&	0	1
0	0	0
1	0	1

|	0	1
0	0	1
1	1	1

^	0	1
0	0	1
1	1	0

Table 2.3: **Operations of Boolean Algebra.** Binary values 1 and 0 encode logic values TRUE and FALSE, while operations `~`, `&`, `|`, and `^` encode logical operations NOT, AND, OR, and EXCLUSIVE-OR, respectively.

are not fully binary compatible. Binary code is seldom portable across different combinations of machine and operating system.

A fundamental concept of computer systems is that programs, from the perspective of the machine, are simply sequences of bytes. The machine has no information about the original source program, except perhaps some auxiliary tables maintained to aid in debugging.

2.1.7 Boolean Algebras and Rings

Since binary values are at the core of how computers encode, store, and manipulate information, a rich body of mathematical knowledge has evolved around the study of the values 0 and 1. This started with the work of George Boole in the late 19th century, and hence goes under the heading of *Boolean Algebra*. Boole observed that by encoding logic values TRUE and FALSE as binary values 1 and 0, he could formulate an algebra that captures the properties of propositional logic.

Table 2.3 defines several operations in Boolean algebra. Our symbols for representing these operations are chosen to match those used by the C bit-level operations, as will be discussed later. Boolean operation `~` corresponds to the logical operation NOT, denoted in propositional logic as $\neg$. That is, we say that $\neg P$ is true when P is not true, and vice-versa. Correspondingly, `~`p equals 1 when p equals 0, and vice-versa. Boolean operation `&` corresponds to the logical operation AND, denoted in propositional logic as $\wedge$. We say that $P \wedge Q$ holds when both P and Q are true. Correspondingly, p `&` q equals 1 only when $p = q = 1$. Boolean operation `|` corresponds to the logical operation OR, denoted in propositional logic as $\vee$. We say that $P \vee Q$ holds when either P or Q are true. Correspondingly, p `|` q equals 1 when either $p = 1$ or $q = 1$. Boolean operation `^` corresponds to the logical operation EXCLUSIVE-OR, denoted in propositional logic as $\oplus$. We say that $P \oplus Q$ holds when either P or Q are true, but not both. Correspondingly, p `^` q equals 1 when either $p = 1$ and $q = 0$, or $p = 0$ and $q = 1$.

Claude Shannon, who would later found the field of information theory, first made the connection between Boolean algebra and digital logic. In his 1937 Master's thesis, he showed that Boolean algebra could be applied to the design and analysis of networks of electromechanical relays. Although computer technology has advanced considerably since that time, Boolean algebra still plays a central role in digital systems design and analysis.

There are many parallels between integer arithmetic and Boolean algebra, as well as several important differences. In particular, the set of integers, denoted $\mathcal{Z}$, forms a mathematical structure known as a *ring*, denoted $\langle \mathcal{Z}, +, \times, -, 0, 1 \rangle$, with addition serving as the *sum* operation, multiplication as the *product* op-

Shared Properties

Property	Integer Ring	Boolean Algebra
Commutativity	$a + b = b + a$ $a \times b = b \times a$	$a \mid b = b \mid a$ $a \,\&\, b = b \,\&\, a$
Associativity	$(a + b) + c = a + (b + c)$ $(a \times b) \times c = a \times (b \times c)$	$(a \mid b) \mid c = a \mid (b \mid c)$ $(a \,\&\, b) \,\&\, c = a \,\&\, (b \,\&\, c)$
Distributivity	$a \times (b + c) = (a \times b) + (a \times c)$	$a \,\&\, (b \mid c) = (a \,\&\, b) \mid (a \,\&\, c)$
Identities	$a + 0 = a$ $a \times 1 = a$	$a \mid 0 = a$ $a \,\&\, 1 = a$
Annihilator	$a \times 0 = 0$	$a \,\&\, 0 = 0$
Cancellation	$-(-a)) = a$	$\sim(\sim a) = a$

Unique to Rings

Inverse	$a + -a = 0$	—

Unique to Boolean Algebras

Distributivity	—	$a \mid (b \,\&\, c) = (a \mid b) \,\&\, (a \mid c)$
Complement	—	$a \mid \sim a = 1$
Idempotency	— —	$a \,\&\, a = a$ $a \mid a = a$
Absorption	— —	$a \mid (a \,\&\, b) = a$ $a \,\&\, (a \mid b) = a$
DeMorgan's Laws	— —	$\sim(a \,\&\, b) = \sim a \mid \sim b$ $\sim(a \mid b) = \sim a \,\&\, \sim b$

Table 2.4: **Comparison of Integer Ring and Boolean Algebra.** The two mathematical structures share many properties, but there are key differences, particularly between $-$ and $\sim$.

eration, negation as the additive inverse, and elements 0 and 1 serving as the additive and multiplicative identities. The Boolean algebra $\langle\{0,1\}, |, \&, \texttt{\~{}}, 0, 1\rangle$ has similar properties. Table 2.4 highlights properties of these two structures, showing the properties that are common to both and those that are unique to one or the other. One important difference is that $\texttt{\~{}}a$ is not an inverse for a under $|$.

If we replace the OR operation of Boolean algebra by the EXCLUSIVE-OR operation, and the complement operation $\texttt{\~{}}$ with the identity operation I (i.e., $I(a) \doteq a$), we have a structure $\langle\{0,1\}, \hat{}, \&, I, 0, 1\rangle$. This structure is no longer a Boolean algebra—in fact it's a ring. It can be seen to be a particularly simple form of the ring consisting of all integers $\{0, 1, \ldots, K-1\}$ with both addition and multiplication performed modulo K. In this case, we have $K = 2$. That is, the Boolean AND and EXCLUSIVE-OR operations correspond to multiplication and addition modulo 2, respectively. One curious property of this algebra is that every element is its own additive inverse: $a \;\hat{}\; I(a) = a \;\hat{}\; a = 0$.

We can extend the four Boolean operations to also operate on bit vectors, i.e., strings of 0's and 1's of some fixed length w. We define the operations over bit vectors according the their applications to the matching elements of the arguments. For example, we define $[a_{w-1}, a_{w-2}, \ldots, a_0] \& [b_{w-1}, b_{w-2}, \ldots, b_0]$ to be $[a_{w-1} \& b_{w-1}, a_{w-2} \& b_{w-2}, \ldots, a_0 \& b_0]$, and similarly for operations $\texttt{\~{}}$, $\&$, and $\hat{}$. Letting $\{0,1\}^w$ denote the set of all strings of 0's and 1's having length w, and a^w denote the string consisting of w repetitions of symbol a, then one can see that the resulting algebras: $\langle\{0,1\}^w, |, \&, \texttt{\~{}}, 0^w, 1^w\rangle$ and $\langle\{0,1\}^w, \hat{}, \&, I, 0^w, 1^w\rangle$ form Boolean algebras and rings, respectively.

Problem 2.6 [Category 1]:
Fill in the following table showing the results of evaluating Boolean operations on bit vectors

Operation	Result
a	[01101001]
b	[01010101]
$\texttt{\~{}}a$	
$\texttt{\~{}}b$	
$a \& b$	
$a \| b$	
$a \;\hat{}\; b$	

One useful application of bit-vectors is to represent sets of some fixed length w. 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 $\texttt{\~{}}$ 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 $\langle\mathcal{P}(S), \cup, \cap, \emptyset, S\rangle$ forms a Boolean algebra, where $\mathcal{P}(S)$ denotes the set of all subsets of S. 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 and Logical 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, i.e., one declared as type `char`, `short int`, `int`, or `long int`. 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.

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 1's 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 1's, regardless of the word size of the machine. Although the same mask can be written `0x98FDECBA` for a 32-bit machine, such code is not as portable.

Problem 2.7 [Category 2]:
Write C expressions for the following values, with the results for `x` = `0x98FDECBA` 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 match the English language description on a machine with arbitrary word size.

Problem 2.8 [Category 2]:
Write a C expression that will yield a word consisting of the least significant byte of `x`, and the remaining bytes of `y`. For operands `x` = `0x89ABCDEF` and `y` = `0x76543210`, this would give `0x765432EF`.

C also provides a set of *logical* operators: `||`, `&&`, and `!` that 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.

Problem 2.9 [Category 2]:
Using only bit-level and logical operations, write C expressions that yield 1 for the described condition and 0 otherwise. Your code should work on a machine with any word size.

A. Any bit of `x` equals 1.

B. Any bit of `x` equals 0.

C. Any bit in the least significant byte of `x` equals 1.

D. Any bit in the least significant byte of `x` equals 0.

Problem 2.10 [Category 2]:
Using only bit-level and logical operations, write a C expression that is equivalent to `x == y`.

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` will yield 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 0's. 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 0's, 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 is the case.

Problem 2.11 [Category 3]:
Write a procedure `int_shifts_are_arithmetic()` that yields 1 when run a machine that uses arithmetic right shifts for `int`'s and 0 otherwise. Your code should work on a machine with any word size. Test your code on several machines. Write and test a procedure `unsigned_shifts_are_arithmetic()` that determines the form of shifts used for `unsigned int`'s.

Problem 2.12 [Category 2]:
We are given the task of writing a procedure `int_size_is_32()` that yields 1 when run on a machine for which an `int` is 32 bits, and yields 0 otherwise. Here is our first attempt:

```
int int_size_is_32()
{
  /* Set most significant bit (msb) of 32-bit word */
  int set_msb = 1 << 31;
  /* Shift past msb of 32-bit word */
  int beyond_msb = 1 << 32;
  /* set_msb is nonzero when size >= 32
     beyond_msb is zero when size <= 32  */
  return set_msb && !beyond_msb;
}
```

When compiled an run on a 32-bit SUN SPARC, however, this procedure returns 0. The following compiler message gives us an indication of the problem:

```
warning: left shift count >= width of type
```

A. In what way does our code fail to comply with the C standard?

B. Modify the code to run properly on any machine for which `int`'s are at least 32 bits.

C. Modify the code to run properly on any machine for which `int`'s are at least 16 bits.

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	$4,294,967,295$

Table 2.5: **C Integral Data types.** Text in square brackets is optional.

Problem 2.13 [Category 2]:
To show how the ring properties of ˆ can be useful, consider the following program:

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

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.
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, i.e., $a \mathbin{\hat{}} a = 0$.

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

2.2 Integer Representations

2.2.1 Integral Data Types

C supports a variety of *integral* data types, as indicated in Table 2.5. 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 Table 2.2. As indicated in Table 2.5, 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 table, 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 Table 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.

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 $\vec{x}$:

$$B2U_w(\vec{x}) \doteq \sum_{i=0}^{w-1} x_i 2^i \tag{2.1}$$

That is, function $B2U_w$ maps length w strings of 0's and 1's to nonnegative integers. The least value is given by bit vector $[00 \cdots 0]$ having integer value $UMin_w \doteq 0$, and the greatest value is given by bit vector $[11 \cdots 1]$ having integer value $UMax_w \doteq \sum_{i=0}^{w-1} 2^i = 2^w - 1$. Thus, the function$B2U_w$ can be defined as a mapping $B2U_w\colon \{0,1\}^w \rightarrow \{0, \ldots, 2^w - 1\}$. Note that $B2U_w$ is a *bijection*, i.e., it associates a unique value to each length w bit vector, and conversely each integer between 0 and $2^w - 1$ has a unique binary representation.

For many applications, we wish to represent nonnegative values as well. The most common computer representation of signed numbers is known as *two's complement* form. This is defined by interpreting the most significant bit of the word to have negative weight:

$$B2T_w(\vec{x}) \doteq -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \tag{2.2}$$

The most significant bit is also called the *sign bit*. When set to 1, the represented value is negative, and when set to 0 the value is nonnegative. The least representable value is given by bit vector $[10 \cdots 0]$ (i.e., set the bit with negative weight but clear all others) having integer value $TMin_w \doteq -2^{w-1}$. The greatest value is given by bit vector $[01 \cdots 1]$, having integer value $TMax_w \doteq \sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$. Again, one can see that $B2T_w$ is a bijection $B2T_w\colon \{0,1\}^w \rightarrow \{-2^{w-1}, \ldots, 2^{w-1} - 1\}$, associating a unique integer in the representable range for each bit pattern.

Table 2.6 shows the bit patterns and numeric values for several "interesting" numbers for different word sizes. The first three give the ranges of representable integers. A few points are worth highlighting. First,

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

Table 2.6: **"Interesting" Numbers.** Both numeric values and hexadecimal representations are shown.

the two's complement range is asymmetric: $|TMin_w| = |TMax_w| + 1$, i.e,. there is no positive counterpart to $TMin_w$. As we shall see, this leads to some peculiar properties of two's complement arithmetic and can be the source of subtle program bugs. Second, the maximum unsigned value is nearly twice the maximum two's complement value: $UMax_w = 2TMax_w + 1$. This follows from the fact that two's complement notation reserves half of the bit patterns to represent negative values. The other cases are the constants -1 and 0. Note that -1 has the same bit representation as $UMax_w$—a string of all 1's. Numeric value 0 is represented as a string of all 0's in both representations.

The C standard does not require signed integers to be represented in two's complement form, but nearly all machines do so. To keep code portable, one should not assume any particular range of representable values or how they are represented, beyond the ranges indicated in Table 2.2. The C library file `<limits.h>` defines a set of constants delimiting the ranges of the different integer data types for the particular machine on which the compiler is running. For example, it defines constants `INT_MAX`, `INT_MIN`, and `UINT_MAX` corresponding to the values of $TMax_w$, $TMin_w$, and $UMax_w$, where w is the size of data type `int`, assuming a two's complement representation.

As an example, consider the following code:

```
short int x = 12345;
short int mx = -x;
show_bytes(&x, sizeof(short int));
show_bytes(&mx, sizeof(short int));
```

When run on a Big Endian machine, this code prints `30 39` and `cf c7`, indicating that `x` has hexadecimal representation `0x3039`, while `mx` has hexadecimal representation `0xCFC7`. Expanding these into binary we get bit patterns [0011000000111001] for `x` and [1100111111000111] for `mx`. As Table 2.7 shows, Equation 2.2 yields values 12345 and -12345 for these two bit patterns.

Weight	12345		−12345		53191	
	Bit	Value	Bit	Value	Bit	Value
1	1	1	1	1	1	1
2	0	0	1	2	1	2
4	0	0	1	4	1	4
8	1	8	0	0	0	0
16	1	16	0	0	0	0
32	1	32	0	0	0	0
64	0	0	1	64	1	64
128	0	0	1	128	1	128
256	0	0	1	256	1	256
512	0	0	1	512	1	512
1024	0	0	1	1024	1	1024
2048	0	0	1	2048	1	2048
4096	1	4096	0	0	0	0
8192	1	8192	0	0	0	0
16384	0	0	1	16384	1	16384
±32768	0	0	1	−32768	1	32768
Total		12345		−12345		53191

Table 2.7: **Two's Complement Representations of** 12345 **and** -12345**, and Unsigned Representation of** 53191**.** -12345 and 53191 have identical bit representations.

2.2.3 Conversions Between Signed and Unsigned

Since both $B2U_w$ and $B2T_w$ are bijections, they have well-defined inverses. Define $U2B_w$ to be $B2U_w^{-1}$, and $T2B_w$ to be $B2T_2^{-1}$. These functions give the unsigned or two's complement bit patterns for a numeric value. Given an integer x in the range $0 \leq x < 2^w$, the function $U2B_w(x)$ gives the unique w-bit unsigned representation of x. Similarly, when x is in the range $-2^{w-1} \leq x < 2^{w-1}$, the function $T2B_w(x)$ gives the unique w-bit two's complement representation of x. Observe that for values in the range $0 \leq x < 2^{w-1}$, both of these functions will yield the same bit representation—the most significant bit will be 0, and hence it does not matter whether this bit has positive or negative weight.

Consider the following function: $U2T_w(x) \doteq B2T_w(U2B_w(x))$. This function takes a number between 0 and $2^{w-1}-1$ and yields a number between -2^{w-1} and $2^{w-1}-1$, where the two numbers have identical bit representations, except that the argument is unsigned, while the result has a two's complement representation. Conversely, the function $T2U_w(x) \doteq B2U_w(T2B_w(x))$ yields the unsigned number having an identical bit representation as the two's complement value of `x`. For example, as Table 2.7 indicates, the 16-bit, two's complement representation of -12345 is identical to the 16-bit, unsigned representation of 53191. Therefore $T2U_{16}(-12345) = 53191$, and $U2T_{16}(53191) = 12345$.

These two functions might seem purely of academic interest, but interestingly, they define the effect of casting between signed and unsigned values in C. Consider the code:

```
int x = -1;
unsigned ux = (unsigned) x;
```

This code will set `ux` to $UMax_w$, where w is the number of bits in data type `int`, since by Table 2.6 we can see that the w-bit two's complement representation of -1 has the same bit representation as $UMax_w$. In general casting from a signed value `x` to unsigned value `(unsigned) x` is equivalent to applying function $T2U$. The cast does not change the bit representation of the argument, just how these bits are interpreted numerically. Similarly, casting from unsigned value `u` to signed value `(int) u` is equivalent to applying function $U2T$.

Comparing Equations 2.1 and 2.2, we can see that for bit pattern $\vec{x}$, if we compute the difference $B2U_w(\vec{x}) - B2T_w(\vec{x})$, the weighted sums for bits from 0 to $w-2$ will cancel each other, leaving a value: $B2U_w(\vec{x}) - B2T_w(\vec{x}) = x_{w-1}(2^{w-1} - -2^{w-1}) = x_{w-1}2^w$. This gives a relationship $B2U_w(\vec{x}) = x_{w-1}2^w + B2T_w(\vec{x})$. If we let $x = B2T_w(\vec{x})$, we then have

$$B2U_w(T2B_w(x)) \quad = \quad T2U_w(x) \quad = \quad x_{w-1}2^w + x \tag{2.3}$$

This relationship is useful for proving relationships between unsigned and two's complement arithmetic. In the two's complement representation of x, bit x_{w-1} determines whether or not x is negative, giving

$$T2U_w(x) \quad = \quad \begin{cases} x + 2^w, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{2.4}$$

On the other hand, if we let $x = B2U_w(\vec{x})$, we have

$$B2T_w(U2B_w(x)) \quad = \quad U2T_w(x) \quad = \quad -x_{w-1}2^w + x \tag{2.5}$$

In the unsigned representation of x, bit x_{w-1} determines whether or not x is greater than or equal to 2^{w-1}, giving

$$U2T_w(x) = \begin{cases} x, & x < 2^{w-1} \\ x - 2^w, & x \geq 2^{w-1} \end{cases} \tag{2.6}$$

For values in the range $0 \leq x < 2^{w-1}$ we have $T2U_w(x) = x$ and $U2T_w(x) = x$. That is, numbers in this range have identical unsigned and two's complement representations. 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 0 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 Table 2.7, we can see that $T2U_{16}(-12345) = 65536 + -12345 = 53191$.

2.2.4 Signed vs. Unsigned in C

As indicated in Table 2.5, C supports both signed and unsigned arithmetic for all of its integer data types. Although the C standard does not specify 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. To create an unsigned constant, the character '`U`' or '`u`' must be added as suffix, e.g., `12345U` or `0x1A2Bu`.

C allows conversion between unsigned and signed. The rule is that the underlying bit representation is not changed. Thus, on a two's complement machine, the effect is to apply the function $U2T_w$ when converting from unsigned to signed, and $T2U_w$ when converting from signed to unsigned, where w is the number of bits for the data type.

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

```
int tx, ty;
unsigned ux, uy;
tx = (int) ux;
uy = (unsigned) ty;
```

or implicitly when an expression of one type is assigned to a variable of another, e.g.,

```
int tx, ty;
unsigned ux, uy;
tx = ux; /* Cast to signed */
uy = ty; /* Cast to unsigned */
```

When printing numeric values with `printf`, the directives `%d`, `%u`, and `%x` should be 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 type information, and so it is possible to print a value of type `int` with directive `%u` and a value of type `unsigned` with directive `%d`. For example, consider the following code:

Expression	Type	Evaluation
`0 == 0U`	unsigned	1
`-1 < 0`	signed	1
`-1 < 0U`	unsigned	0 *
`2147483647 > -2147483648`	signed	1
`2147483647U > -2147483648`	unsigned	0 *
`2147483647 > (int) 2147483648U`	signed	1 *
`-1 > -2`	signed	1
`(unsigned) -1 > -2`	unsigned	0 *

Table 2.8: **Effects of C Promotion Rules on 32-Bit Machine.** Nonintuitive cases marked by '*'. When either operand of a comparison is unsigned, the other operand is implicitly cast to unsigned.

```
int x = -1;
unsigned u = 2147483648; /* 2^(31) */
printf("x = %u = %d\n", x, x);
printf("u = %u = %d\n", u, u);
```

When run on a 32-bit machine it prints the following:

```
x = 4294967295 = -1
u = 2147483648 = -2147483648
```

In both cases, `printf` prints the word first as if it represented an unsigned number and second as if it represented a signed number. We can see the conversion routines in action: $T2U_{32}(-1) = UMax_{32} = 4,294,967,295$ and $U2T_{32}(2^{31}) = 2^{31} - 2^{32} = -2^{31} = TMax_{32}$.

Some peculiar behavior arises due to C's handling of expressions containing combinations of signed and unsigned quantities. When an operation is performed where one operand is signed and the other is unsigned, C implicitly casts the signed argument to unsigned and performs the operations assuming the numbers are nonnegative. As we will see, this convention makes little difference for standard arithmetic operations, but it leads to nonintuitive results for relational operators such as < and >. Table 2.8 shows some example relational expressions and their resulting evaluations, assuming a 32-bit machine using two's complement representation. The nonintuitive cases are marked by '*'. Consider the comparison `-1 < 0U`. Since the second operand is unsigned, the first one is implicitly cast to unsigned, and hence the expression is equivalent to the comparison `4294967295U < 0U` (recall that $T2U_w(-1) = UMax_w$), which of course is false. The other cases can be understood by similar analyses.

2.2.5 Sign Extension

One common operation is to convert between integers having different word sizes, while retaining the same numeric value. Of course, this may not be possible when the destination data type is too small to represent the desired value. Converting from a smaller to a larger data type, however, should always be possible. To convert an unsigned number to a larger data type, we can simply add leading 0's to the representation. 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. Thus, if our original value has bit representation $[x_{w-1}, x_{w-2}, \ldots, x_0]$, the expanded representation would be $[x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]$.

As an example, consider the following code:

```
short           sx = -12345;
unsigned short usx = sx;   /*  53191 */
int              x = sx;   /* -12345 */
unsigned        ux = usx;  /*  53191 */
printf("sx = %d: ", sx);
show_bytes((byte_pointer) &sx, sizeof(short));
printf("usx = %u: ", usx);
show_bytes((byte_pointer) &usx, sizeof(unsigned short));
printf("x = %d: ", x);
show_bytes((byte_pointer) &x, sizeof(int));
printf("ux = %u: ", ux);
show_bytes((byte_pointer) &ux, sizeof(unsigned));
```

When run on a 32-bit, big endian machine using two's complement representations this code prints:

```
sx = -12345:  cf c7
usx = 53191:  cf c7
x = -12345:  ff ff cf c7
ux = 53191:  00 00 cf c7
```

We see that although the two's complement representation of -12345 and the unsigned representation of 53191 are identical for a 16-bit word size, they differ for a 32-bit word size. In particular, -12345 has hexadecimal representation `0xFFFFCFC7`, while 53191 has hexadecimal representation `0x0000CFC7`. The former has been sign-extended—16 copies of the most significant bit 1, having hexadecimal representation `0xFFFF` have been added as leading bits. The latter has been padded with 16 leading 0's, having hexadecimal representation `0x0000`.

Can we justify that sign extension works? What we want to prove is that

$$B2T_{w+k}([x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) \quad = \quad B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])$$

where in the expression on the left-hand side, we have made k additional copies of bit x_{w-1}. The proof follows by induction on k. That is, if we can prove that sign-extending by one bit preserves the numeric value, then this property will hold when sign-extending by an arbitrary number of bits. Thus, the task reduces to proving that

$$B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) \quad = \quad B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])$$

Expanding the left-hand expression with Equation 2.2 gives

$$
\begin{aligned}
B2T_{w+1}([x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_0]) &= -x_{w-1}2^w + \sum_{i=0}^{w-1} x_i 2^i \\
&= -x_{w-1}2^w + x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\
&= -x_{w-1}(2^w - 2^{w-1}) + \sum_{i=0}^{w-2} x_i 2^i \\
&= -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i \\
&= B2T_w([x_{w-1}, x_{w-2}, \ldots, x_0])
\end{aligned}
$$

The key property we exploit is that $-2^w + 2^{w-1} = -2^{w-1}$. Thus, the combined effect of adding a bit of weight -2^w and of converting the bit having weight -2^{w-1} to be one with weight 2^{w-1} is to preserve the original numeric value.

One point worth making is that the relative order of conversion from one data size to another and between unsigned and signed can affect the behavior of a program. Consider the following additional code for our previous example:

```
unsigned  uy = x;   /* Mystery! */
printf("uy = %u: ", uy);
show_bytes((byte_pointer) &uy, sizeof(unsigned));
```

This portion of the code causes the following to be printed:

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

This shows that the expressions:

```
(unsigned) (int) sx
```

and

```
(unsigned) (unsigned short) sx
```

produce different values, even though the original and the final data types are the same. In the former expression, we first sign extend the 16-bit `short` to a 32-bit `int`, whereas no sign extension is performed in the latter expression.

2.2.6 Truncating Numbers

Suppose that rather than padding a value with extra bits, we truncate a number to have fewer bits. This occurs for example in the code:

```
int    x = 53191;
short sx = (short) x; /* -12345 */
int    y = sx;        /* -12345 */
```

On a typical 32-bit machine, when we cast `x` to be `short`, we truncate the 32-bit `int` to be a 16-bit `short int`. As we saw before, this 16 bit pattern is the two's complement representation of -12345. When we cast this back to `int`, sign extension will set the high order 16 bits to 1's, yielding the 32-bit two's complement representation of -12345.

In general, 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 $\vec{x}' = [x_{k-1}, x_{k-2}, \ldots, x_0]$.

For an unsigned number x, the result of truncating it to k bits is equivalent to computing $x \bmod 2^k$. This can be seen by applying the modulus operation to Equation 2.1:

$$
\begin{aligned}
B2U_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k &= \left[\sum_{i=0}^{w-1} x_i 2^i\right] \bmod 2^k \\
&= \left[\sum_{i=0}^{k-1} x_i 2^i\right] \bmod 2^k \\
&= \sum_{i=0}^{k-1} x_i 2^i \\
&= B2U_k([x_k, x_{k-1}, \ldots, x_0])
\end{aligned}
$$

In the above derivation we make use of the property that $2^i \bmod 2^k = 0$ for any $i \geq k$, and that $\sum_{i=0}^{k-1} x_i 2^i \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1 < 2^k$.

For a two's complement number x, a similar argument shows that $B2T_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k = B2U_k([x_k, x_{k-1}, \ldots, x_0])$. That is, $x \bmod 2^k$ can be represented by an unsigned number having bit-level representation $[x_{k-1}, \ldots, x_0]$. In general, however, we treat the truncated number as being signed. This will have numeric value $U2T_k(x \bmod 2^k)$.

Summarizing, the effect of truncation can be seen to be:

$$
\begin{aligned}
B2U_k([x_k, x_{k-1}, \ldots, x_0]) &= B2U_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k && (2.7) \\
B2T_k([x_k, x_{k-1}, \ldots, x_0]) &= U2T_k(B2T_w([x_w, x_{w-1}, \ldots, x_0]) \bmod 2^k) && (2.8)
\end{aligned}
$$

2.2.7 Advice on Signed vs. Unsigned

As we have seen, the implicit casting of signed to unsigned leads to some nonintuitive behavior. Nonintuitive features often lead to program bugs, and ones involving the nuances of implicit casting can be especially difficult to see.

Problem 2.14 [Category 2]:
Consider the following code that (attempts to) sum the elements in an array `a`, where the number of elements is given by parameter `length`:

```
int sum_elements(float a[], unsigned length) {
{
  int i;
  float result = 0;
  for (i = 0; i <= length-1; i++)
    result += a[i];
  return result;
}
```

When run with argument `length` equal to 0, this code should return 0.0. Instead it encounters a memory error. Explain why this happens. Show how this code can be corrected.

One way to avoid such bugs is to never use unsigned numbers. In fact, few languages other than C support unsigned integers. Apparently these other language designers viewed them as more trouble than they are worth.

Unsigned values are very useful when we want to think of words as just collections of bits with no numeric interpretation. This occurs for example when want to pack a word with *flags* describing various Boolean conditions. Unsigned values are also useful when implementing mathematical packages for modular arithmetic and for multi-precision arithmetic, in which numbers are represented by arrays of words.

2.3 Integer Arithmetic

Many beginning programmers are surprised to find that adding two positive numbers can yield a negative result, and that the comparison `x < y` can yield a different result than the comparison `x - y < 0`. These properties are artifacts of the finite nature of computer arithmetic. Understanding the nuances of computer arithmetic can help programmers write more reliable code.

2.3.1 Unsigned Addition

Consider two nonnegative integers x and y such that $0 \leq x, y \leq 2^w - 1$. Each of these numbers can be represented by w-bit unsigned numbers. If we compute their sum, however, we have a possible range $0 \leq x + y \leq 2^w - 2$. Representing this sum could require $w + 1$ bits. Furthermore, if we add this sum to another value, we may require $w + 2$ bits, and so on. This continued "word size inflation" means we cannot place any bound on the word size required to fully represent the results of arithmetic operations. Some programming languages, such as Lisp, actually support *infinite precision* arithmetic to allow arbitrary (within the memory limits of the machine, of course) integer arithmetic. More commonly, programming languages support fixed-precision arithmetic, and hence operations such as "addition" and "multiplication" differ from their counterpart operations over integers.

Unsigned arithmetic can be viewed as a form of modular arithmetic. Unsigned addition is equivalent to computing the sum modulo 2^w. This value can be computed by simply discarding the high order bit in the $w+1$-bit representation of $x+y$. For example, consider a 4-bit number representation with $x = 9$ and $y = 12$, having bit representations $[1001]$ and $[1100]$, respectively. Their sum is 21, having a 5-bit representation $[10101]$. But if we discard the high order bit we get $[0101]$, i.e., decimal value 5. This matches the value $21 \bmod 16 = 5$.

In general, we can see that if $x+y < 2^w$, the leading bit in the $w+1$-bit representation of the sum will equal 0, and hence discarding it will not change the numeric value. On the other hand, if $2^w \leq x+y < 2^{w+1}$, the leading bit in the $w+1$-bit representation of the sum will equal 1, and hence discarding it is equivalent to subtracting 2^w from the sum. This will give us a value in the range $0 \leq x+y-2^w < 2^{w+1} - 2^w = 2^w$, which is precisely the modulo 2^w sum of x and y. Let us define the operation $+^{u}_{w}$ for arguments x and y such that $0 \leq x, y < 2^w$ as:

$$x +^{u}_{w} y \quad = \quad \begin{cases} x+y, & x+y < 2^w \\ x+y-2^w, & 2^w \leq x+y < 2^{w+1} \end{cases} \tag{2.9}$$

This is precisely the result we get in C when performing addition on two w-bit unsigned values.

An arithmetic operation is said to *overflow* when the full integer result cannot fit within the word size limits of the data type. When executing C programs, such overflows are not signalled as errors. At times, however, we might wish to determine whether overflow has occurred. For example, suppose we compute $s \doteq x +^{u}_{w} y$, and we wish to determine whether s equals $x+y$. We claim that overflow has occurred if and only if $s < x$ (or equivalently $s < y$.) To see this, observe that $x+y \geq x$, and hence if s did not overflow, we will surely have $s \geq x$. 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 $s = x+y-2^w < x$. In our earlier example, we saw that $9 +^{u}_{4} 12 = 5$. We can see that overflow occurred, since $5 < 9$.

Modular addition forms a mathematical structure known as an *Abelian group*. That is, it is commutative (that's where the "Abelian" part comes in) and associative. It has an identity element 0, and every element has an additive inverse. Let us consider the set of w-bit unsigned numbers with addition operation $+^{u}_{w}$. For every value x, there must be some value $-^{u}_{w}\, x$ such that $-^{u}_{w}\, x +^{u}_{w} x = 0$. 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 $0 \leq 2^w - x < 2^w$, and $(x + 2^w - x) \bmod 2^w = 2^w \bmod 2^w = 0$. Hence it is the inverse of x under $+^{u}_{w}$. These two cases lead to the following equation for $0 \leq x < 2^w$:

$$-^{u}_{w}\, x \quad = \quad \begin{cases} x, & x = 0 \\ 2^w - x, & x > 0 \end{cases} \tag{2.10}$$

2.3.2 Two's Complement Addition

A similar problem arises for two's complement addition. Given integer values x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$, their sum is in the range $-2^w \leq x+y \leq 2^w - 2$, potentially requiring $w+1$-bits to represent exactly. As before, we avoid ever-expanding data sizes by truncating the representation to w bits. The result is not as familiar mathematically as modular addition, however.

The w-bit two's complement sum of two numbers has the exact same bit-level representation as the unsigned sum. In fact, most computers use the same machine instruction to perform either unsigned or signed addition. Thus, we can define two's complement addition for word size w, denoted as $+^t_w$ on operands x and y such that $-2^w \leq x, y \leq 2^w - 2$ as

$$x +^t_w y \ \doteq \ \mathit{U2T}_w(\mathit{T2U}_w(x) +^u_w \mathit{T2U}_w(y)) \tag{2.11}$$

By Equation 2.3 we can write $\mathit{T2U}_w(x)$ as $-x_{w-1}2^w + x$, and $\mathit{T2U}_w(y)$ as $-y_{w-1}2^w + y$. Using the property that $+^u_w$ is simply addition modulo 2^w, along with the properties of modular addition, we then have

$$\begin{aligned} x +^t_w y &= \mathit{U2T}_w(\mathit{T2U}_w(x) +^u_w \mathit{T2U}_w(y)) \\ &\quad \mathit{U2T}_w[(-x_{w-1}2^w + x + -y_{w-1}2^w + y) \bmod 2^w] \\ &\quad \mathit{U2T}_w[(x+y) \bmod 2^w] \end{aligned}$$

The terms $-x_{w-1}2^w$ and $-y_{w-1}2^w$ drop out since they equal 0 modulo 2^w.

To better understand this quantity, let us define z as the integer sum $z \doteq x + y$, z' as $z' \doteq z \bmod 2^w$, and z'' as $z'' \doteq \mathit{B2T}_w(z')$. We can divide the analysis into 4 cases:

1. $-2^w \leq z < -2^{w-1}$. Then we will have $z' = z + 2^w$. This gives $0 \leq z' < -2^{w-1} + 2^w = 2^{w-1}$. Examining Equation 2.6, we see that z' is in the range such that $z'' = z'$. This case is referred to as *negative overflow*. We have added two negative numbers x and y (that's the only way we can have $z < -2^{w-1}$) and obtained a nonnegative result $z'' = x + y + 2^w$.

2. $-2^{w-1} \leq z < 0$. Then we will again have $z' = z + 2^w$, giving $-2^{w-1} + 2^w = 2^{w-1} \leq z' < 2^w$. Examining Equation 2.6, we see that z' is in such a range that $z'' = z' - 2^w$, and therefore $z'' = z' - 2^w = z + 2^w - 2^w = z$. That is, our two's complement sum z'' equals the integer sum $x + y$.

3. $0 \leq z < 2^{w-1}$. Then we will have $z' = z$, giving $0 \leq z' < 2^{w-1}$, and hence $z'' = z' = z$. Again, the two's complement sum z'' equals the integer sum $x + y$.

4. $2^{w-1} \leq z < 2^w$. We will again have $z' = z$, giving $2^{w-1} \leq z' < 2^w$. But in this range we have $z'' = z' - 2^w$, giving $z'' = x + y - 2^w$. This case is referred to as *positive overflow*. We have added two positive numbers x and y (that's the only way we can have $z \geq 2^{w-1}$) and obtained a negative result $z'' = x + y - 2^w$.

By the preceding analysis, we have shown that when operation $+^t_w$ is applied to values x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$, we have

$$x +^t_w y \ = \ \begin{cases} x + y - 2^w, & 2^{w-1} \leq x + y \qquad \text{Positive Overflow} \\ x + y, & -2^{w-1} \leq x + y < 2^{w-1} \\ x + y + 2^w, & x + y < -2^{w-1} \qquad \text{Negative Overflow} \end{cases} \tag{2.12}$$

As an illustration, Table 2.9 shows some examples of 4-bit two's complement addition. Each example is labeled by the case to which it corresponds in the derivation of Equation 2.12. Note that $2^4 = 16$, and hence negative overflow yields a result 16 more than the integer sum, and positive overflow yields a result 16 less. We include bit-level representations of the operands and the result. Observe that the result can be obtained by performing binary addition of the operands and truncating the result to 4 bits.

x	y	$x+y$	$x +^t_4 y$	Case
-8 [1000]	-5 [1011]	-13	3 [0011]	1
-8 [1000]	-8 [1000]	-16	0 [0000]	1
-8 [1000]	5 [0101]	-3	-3 [1101]	2
2 [0010]	5 [0101]	7	7 [0111]	3
5 [0101]	5 [0101]	10	-6 [1010]	4

Table 2.9: **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.

Problem 2.15 [Category 1]:
Fill in the following table in the style of Table 2.9. Give the integer values of the 5-bit arguments, the values of both their integer and two's complement sums, the bit-level represntation of the two's complement sum, and the case from the derivation of Equation 2.12.

x	y	$x+y$	$x +^t_4 y$	Case
[10000]	[10101]			
[10000]	[10000]			
[11000]	[00111]			
[11110]	[00101]			
[01000]	[01000]			

Equation 2.12 also lets us identify the cases where overflow has occurred. When both x and y are negative, but $x +^t_w y \geq 0$, we have negative overflow. When both x and y are positive, but $x +^t_w y < 0$, we have positive overflow.

2.3.3 Two's Complement Negation

One can see that every number x in the range $-2^{w-1} \leq x < 2^{w-1}$ has an additive inverse under $+^t_w$ as follows. First, for $x \neq -2^w$, we can see that its additive inverse is simply $-x$. That is, we have $-2^{w-1} < -x < 2^{w-1}$ and $-x +^t_w x = -x + x = 0$. For $x = -2^{w-1} = \mathit{TMin}_w$, on the other hand,

$-x = 2^{w-1}$ cannot be represented as a w-bit number. We claim that this special value has itself as the additive inverse under $+^t_w$. The value of $-2^{w+1} +^t_w -2^{w+1}$ is given by the third case of Equation 2.12, since $-2^{w-1} + -2^{w-1} = -2^w$. This gives $-2^{w+1} +^t_w -2^{w+1} = -2^w + 2^w = 0$. From this analysis we can define the two's complement negation operation $-^t_w$ for x in the range $-2^{2-1} \leq x < 2^{w-1}$ as:

$$-^t_w\, x \quad = \quad \begin{cases} -2^{w-1}, & x = -2^{w-1} \\ -x, & x > -2^{w-1} \end{cases} \tag{2.13}$$

A well-known technique for performing two's complement negation at the bit-level is to complement the bits and then increment the result. In C, this can be written as `~x + 1`. To justify the correctness of this technique, observe that for any single bit x_i, we have $\tilde{}x_i = 1 - x_i$. Let $\vec{x}$ be a bit vector of length w and $x \doteq B2T_w(\vec{x})$ be the two's complement number it represents. By Equation 2.2, the complemented bit vector $\tilde{}\vec{x}$ has numeric value

$$\begin{aligned} B2T_w(\tilde{}\vec{x}) &= -(1 - x_{w-1})2^{w-1} + \sum_{i=0}^{w-2}(1 - x_i)2^i \\ &= \left[-2^{w-1} + \sum_{i=0}^{w-2} 2^i\right] + - \left[-x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i\right] \\ &= [-2^{w-1} + 2^{w-1} - 1] - B2T_w(\vec{x}) \\ &= -1 - x \end{aligned}$$

The key simplification in the above derivation is that $\sum_{i=0}^{w-2} 2^i = 2^{w-1} - 1$. It follows that by incrementing $\tilde{}\vec{x}$ we obtain $-x$.

To increment a number x represented at the bit-level as $\vec{x} \doteq [x_{w-1}, x_{w-2}, \ldots, x_0]$, define the operation $incr$ as follows. Let k be the position of the rightmost zero, i.e., such that $\vec{x}$ is of the form $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 0, 1, \ldots, 1]$. We then define $incr(\vec{x})$ to be $[x_{w-1}, x_{w-2}, \ldots, x_{k+1}, 1, 0, \ldots, 0]$. For the special case where the bit-level representation of x is $[1, 1, \ldots, 1]$, define $incr(\vec{x})$ to be $[0, \ldots, 0]$. To show that $incr(\vec{x})$ yields the bit-level representation of $x +^t_w 1$, consider the following cases:

1. When $\vec{x} = [1, 1, \ldots, 1]$, we have $x = -1$. The incremented value $incr(\vec{x}) \doteq [0, \ldots, 0]$ has numeric value 0.

2. When $k = w - 1$, i.e., $\vec{x} = [0, 1, \ldots, 1]$, we have $x = TMax_w$. The incremented value $incr(\vec{x}) = [1, 0, \ldots, 0]$ has numeric value $TMin_w$. From Equation 2.12, we can see that $TMax_w +^t_w 1$ is one of the positive overflow cases, yielding $TMin_w$.

3. When $k < w - 1$, i.e., $x \neq TMax_w$ and $x \neq -1$, we can see that the low-order $k + 1$ bits of $incr(\vec{x})$ has numeric value 2^k, while the low-order $k + 1$ bits of $\vec{x}$ has numeric value $\sum_{i=0}^{k-1} 2^i = 2^k - 1$. The high-order $w - k + 1$ bits have matching numeric values. Thus, $incr(\vec{x})$ has numeric value $x + 1$. For all cases other than $x = TMax_w$, adding 1 to x will not cause any overflow, and hence $x +^t_w 1 = x + 1$.

As illustrations, Table 2.10 shows how complementing and incrementing affect the numeric values of several 4-bit vectors.

$\vec{x}$		$\tilde{~}\vec{x}$		$incr(\tilde{~}\vec{x})$	
[0101]	5	[1010]	−6	[1011]	−5
[0111]	7	[1000]	−8	[1001]	−7
[1100]	−4	[0011]	3	[0100]	4
[0000]	0	[1111]	−1	[0000]	0
[1000]	−8	[0111]	7	[1000]	−8

Table 2.10: **Examples of Complementing and Incrementing 4-bit numbers.** The effect is to compute the two's value negation.

Problem 2.16 [Category 1]:
Fill in the following table showing the effects of complementing and incrementing several 5-bit vectors, in the style of Table 2.10. Show both the bit vectors and the numeric values.

$\vec{x}$	$\tilde{~}\vec{x}$	$incr(\tilde{~}\vec{x})$
[01101]		
[01111]		
[11000]		
[11111]		
[10000]		

Problem 2.17 [Category 2]:
Argue that first decrementing and then complementing is equivalent to complementing and then incrementing. That is, for any signed value `x`, the C expressions `-x`, `~x+1`, and `~(x-1)` yield identical results. What mathematical properties of two's complement addition does your derivation rely on?

2.3.4 Unsigned Multiplication

Integers x and y in the range $0 \leq x, y \leq 2^w - 1$ can be represented as w-bit unsigned numbers, but their product $x \cdot y$ can range between 0 and $(2^w - 1)^2 = 2^{2w} - 2^{w+1} + 1$. This could require as many as $2w$ bits to represent. Instead, unsigned multiplication in C is defined to yield the w-bit value given by the low order w bits of the $2w$-bit integer product. By Equation 2.7, this can be seen to be equivalent to computing the product modulo 2^w. Thus, the effect of the w-bit unsigned multiplication operation $*^u_w$ is:

$$x *^u_w y \quad = (x \cdot y) \bmod 2^w \tag{2.14}$$

It is well known that modular arithmetic forms a ring. We can therefore deduce that unsigned arithmetic over w-bit numbers forms a ring $\langle \{0, \ldots, 2^w - 1\}, +^u_w, *^u_w, -^u_w, 0, 1 \rangle$.

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

Table 2.11: **3-Bit Unsigned and Two's Complement Multiplication Examples.** Although the bit-level representations of the full products may differ, those of the truncated products are identical.

2.3.5 Two's Complement Multiplication

Integers x and y in the range $-2^{w-1} \leq x, y \leq 2^{w-1} - 1$ can be represented as w-bit two's complement numbers, but their product $x \cdot y$ can range between $-2^{w-1} \cdot (2^{w-1} - 1) = -2^{2w-2} + 2^{w-1}$ and $-2^{w-1} \cdot -2^{w-1} = 2^{2w-2}$. This could require as many as $2w$ bits to represent in two's complement form—most cases would fit into $2w - 1$ bits, but the special case of 2^{2w-2} requires the full $2w$ bits. Instead, signed multiplication in C is generally performed by truncating the $2w$-bit product to w bits. By Equation 2.8, the effect of the w-bit two's complement multiplication operation $*^t_w$ is:

$$x *^t_w y \quad = \quad U2T_w((x \cdot y) \bmod 2^w) \tag{2.15}$$

We claim that the bit-level representation of the product operation is identical for both unsigned and two's complement multiplication. That is, given bit vectors $\vec{x}$ and $\vec{y}$ of length w, the bit-level representation of the unsigned product $B2U_w(\vec{x}) *^u_w B2U_w(\vec{y})$ is identical to the bit-level representation of the two's complement product $B2T_w(\vec{x}) *^t_w B2T_w(\vec{x})$. This implies that the machine can use a single type of multiply instruction to multiply both signed and unsigned integers.

To see this, let $x = B2T_w(\vec{x})$ and $y = B2T_w(\vec{y})$ be the two's complement values denoted by these bit patterns, and let $x' = B2U_w(\vec{x})$ and $y' = B2U_w(\vec{y})$ be the unsigned values. From Equation 2.3, we have $x' = x + x_{w-1}2^w$, and $y' = y + y_{w-1}2^w$. Computing the product of these values modulo 2^w gives:

$$\begin{aligned}
(x' \cdot y') \bmod 2^k \quad &= \quad [(x + x_{w-1}2^w) \cdot (y + y_{w-1}2^w)] \bmod 2^w & (2.16)\\
&= \quad [x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}] \bmod 2^w & (2.17)\\
&= \quad (x \cdot y) \bmod 2^w & (2.18)
\end{aligned}$$

Thus, the low-order w bits of $x \cdot y$ and $x' \cdot y'$ are identical.

As illustrations, Table 2.11 shows the results of multiplying different 3-bit numbers. For each pair of bit-level operands, we perform both unsigned and two's complement multiplication. Note that the unsigned, truncated product always equals $x \cdot y \bmod 8$, and that the bit-level representations of both truncated products are identical.

Problem 2.18 [Category 1]:
Fill in the following table showing the results of multiplying different 3-bit numbers, in the style of Table 2.11

Mode	x	y	$x \cdot y$	Truncated $x \cdot y$
Unsigned	[110]	[010]		
Two's Comp.	[110]	[010]		
Unsigned	[001]	[111]		
Two's Comp.	[001]	[111]		
Unsigned	[111]	[111]		
Two's Comp.	[111]	[111]		

Problem 2.19 [Category 3]:
Suppose we want to compute the complete $2w$-bit representation of $x \cdot y$, where both x and y are unsigned, on a machine for which data type `unsigned` is w bits. The low-order w bits of the product can be computed with the expression `x*y`, so we only require a procedure with prototype

```
unsigned unsigned_high_prod(unsigned x, unsigned y);
```

that computes the high-order w bits of $x \cdot y$ for unsigned variables.
We have access to a library function with prototype:

```
int signed_high_prod(int x, int y);
```

that computes the high order w bits of $x \cdot y$ for the case where x and y are in two's complement form. Write code calling this procedure to implement the function for unsigned arguments. Justify the correctness of your solution.
[Hint:] Look at the relationship between the signed product $x \cdot y$ and the unsigned product $x' \cdot y'$ in the derivation of Equation 2.18.

We can see that unsigned arithmetic and two's complement arithmetic over w-bit numbers are isomorphic—the operations $+^u_w$, $-^u_w$, and $*^u_w$ have the exact same effect at the bit level as do $+^t_w$, $-^t_w$, and $*^t_w$. From this we can deduce that two's complement arithmetic forms a ring $\langle\{-2^{w-1}, \ldots, 2^{w-1}-1\}, +^t_w, *^t_w, -^t_w, 0, 1\rangle$.

2.3.6 Multiplying by Powers of Two

On most machines, the integer multiply instruction is fairly slow—requiring 12 or more clock cycles, whereas other integer operations such as addition, subtraction, bit-level operations, and shifting require only one clock cycle. As a consequence, one important optimization used by compilers is to attempt to replace multiplications by constant factors with combinations of shift and addition operations.

Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$. Then for any $k \geq 0$, we claim the bit-level representation of $x2^k$ is given by $[x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]$, where k 0's have been added to the right. This property can be derived using Equation 2.1:

$$\begin{aligned} B2U_{w+k}([x_{w-1}, x_{w-2}, \ldots, x_0, 0, \ldots, 0]) &= \sum_{i=0}^{w-1} x_i 2^{i+k} \\ &= \left[\sum_{i=0}^{w-1} x_i 2^i\right] \cdot 2^k \\ &= x2^k \end{aligned}$$

For $k < w$, we can truncate the shifted bit vector to be of length w, giving $[x_{w-k-1}, x_{w-k-2}, \ldots, x_0, 0, \ldots, 0]$. By Equation 2.7, this bit-vector has numeric value $x2^k \bmod 2^w = x *^u_w 2^k$. Thus, for unsigned variable `x`, the C expression `x << k` is equivalent to `x * pwr2k`, where `pwr2k` equals $2^{\mathtt{k}}$. In particular, we can compute `pwr2k` as `1U << k`.

By similar reasoning, we can show that for a two's complement number x having bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and any k in the range $0 \leq k < w$, bit pattern $[x_{w-k-1}, \ldots, x_0, 0, \ldots, 0]$ will be the two's complement representation of $x *^t_w 2^k$. Therefore, for signed variable `x` , the C expression `x << k` is equivalent to `x * pwr2k`, where `pwr2k` equals $2^{\mathtt{k}}$.

Problem 2.20 [Category 2]:
Suppose we are given the task of generating code to multiply integer variable `x` by various different constant factors K. To be efficient we want to use only the operations `+`, `-`, and `<<`. For the following values of K, write C expressions to perform the multiplication using at most 3 operations per expression.

A. $K = 5$:

B. $K = 9$:

C. $K = 14$:

D. $K = -56$:

2.3.7 Dividing by Powers of Two

Integer division on most machines is even slower than integer multiplication—requiring 30 or more clock cycles. Dividing by a power of two can also be performed using shift operations, but we use a right shift rather than a left shift.

For $x \geq 0$ and $y > 0$, the result of integer division should be $\lfloor x/y \rfloor$, where for any real number a, $\lfloor a \rfloor$ is defined to be the unique integer a' such that $a' \leq a < a' + 1$. As examples: $\lfloor 3.14 \rfloor = 3$, $\lfloor -3.14 \rfloor = -4$, and $\lfloor 3 \rfloor = 3$.

Let x be the unsigned integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and k be in the range $0 \leq k < w$. Let x' be the unsigned number with $w-k$-bit representation $[x_{w-1}, x_{w-2}, \ldots, x_k]$, and x'' be the unsigned number with k-bit representation $[x_{k-1}, \ldots, x_0]$. We claim that $x' = \lfloor x/2^k \rfloor$. To see this, by Equation 2.1, we have $x = \sum_{i=0}^{w-1} x_i 2^i$, $x' = \sum_{i=k}^{w-k-1} x_i 2^{i-k}$ and $x'' = \sum_{i=0}^{k-1} x_i 2^i$. We can therefore write x as $x = 2^k x' + x''$. Observe that $0 \leq x'' \leq \sum_{i=0}^{k-1} 2^i = 2^k - 1$, and hence $0 \leq x'' < 2^k$, implying that $\lfloor x''/2^k \rfloor = 0$. Therefore $\lfloor x/2^k \rfloor = \lfloor x' + x''/2^k \rfloor = x' + \lfloor x''/2^k \rfloor = x'$.

Observe that performing a right shift of bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$ by k yields bit vector $[0, ..., 0, x_{w-1}, x_{w-2}, \ldots, x_k]$. This bit vector has numeric value x', i.e., logically right shifting an unsigned number by k is equivalent to dividing it by 2^k. Therefore, for unsigned variable `x`, the C expression `x >> k` is equivalent to `x / pwr2k`, where `pwr2k` equals 2^{k}.

Let x be the two's complement integer represented by bit pattern $[x_{w-1}, x_{w-2}, \ldots, x_0]$, and k be in the range $0 \leq k < w$. Let x' be the two's complement number represented by the $w-k$ bits $[x_{w-1}, x_{w-2}, \ldots, x_k]$, and x'' be the *unsigned* number with represented by the k bits $[x_{k-1}, \ldots, x_0]$. By a similar analysis as the unsigned case, we have $x = 2^k x' + x''$, and $0 \leq x'' < 2^k$, giving $x' = \lfloor x/2^k \rfloor$. Furthermore, observe that shifting bit vector $[x_{w-1}, x_{w-2}, \ldots, x_0]$ right *arithmetically* by k yields a bit vector $[x_{w-1}, \ldots, x_{w-1}, x_{w-1}, x_{w-2}, \ldots, x_k]$, which is the sign extension from $w-k$ bits to w bits of $[x_{w-1}, x_{w-2}, \ldots, x_k]$. Thus, this shifted bit vector is the two's complement representation of x'.

For $x < 0$ and $y > 0$, the result of integer division should be $\lceil x/y \rceil$, where for any real number a, $\lceil a \rceil$ is defined to be the unique integer a' such that $a' - 1 < a \leq a'$. That is, integer division should round negative results upward toward zero. For example the C expression `-5/2` yields `-2`. Thus, right shifting a negative number by k is not equivalent to dividing it by 2^k when rounding occurs. For example, the 4-bit representation of -5 is [1011]. If we shift it right by one arithmetically we get [1101], which is the two's complement representation of -3.

We can correct for this rounding by "biasing" the value before shifting. This technique exploits the property that $\lceil x/y \rceil = \lfloor (x + y - 1)/y \rfloor$ for integers x and y such that $y > 0$. Thus, for $x < 0$, if we first add $2^k - 1$ to x before right shifting, we will get a correctly rounded result. This analysis shows that the C expression `(x<0 ? (x + (1<<k)-1) : x) >> k` is equivalent to `x/pwr2k`, where `pwr2k` equals 2^{k}. For example, to divide -5 by 2, we first add bias $2 - 1 = 1$ giving bit pattern [1100]. Right shifting this by one arithmetically gives bit pattern [1110], which is the two's complement representation of -2.

2.3.8 Summary

We have seen that the finite integer arithmetic implemented by C has some peculiar properties. For example, the expression `x * x` can evaluate to a negative number. Nonetheless, both unsigned and two's complement arithmetic satisfies the properties of a ring. This allows compilers to do many optimizations. For example, in replacing the expression `7*x` by `(x<<3)-x`, we make use of the associative, commutative and distributive properties, along with the relationship between shifting and multiplying by powers of two.

We have seen several clever ways to exploit combinations bit-level operations and arithmetic operations. For example, we saw that `~x+1` is equivalent to `-x`. As another example, suppose we want a bit pattern of the form $[0, \ldots, 0, 1, \ldots, 1]$, consisting of $w-k$ 0's followed by k 1's. Such bit patterns are useful for masking operations. This such a pattern can be generated by the C expression `(1<<k)-1`, exploiting the property that the desired bit pattern has numeric value $2^k - 1$.

Problem 2.21 [Category 2]:
Write C expressions to generate the following bit patterns, where a^k represents k repetitions of symbol a. Assume a w-bit data type.

A. $1^{w-k}0^k$.

B. $0^{w-k-j}1^k0^j$.

Problem 2.22 [Category 2]:
Suppose we number the bytes in a w-bit word from 0 (least significant) to $w/8 - 1$ (most significant). Write code for the following C function, that will return an unsigned value in which byte `i` of argument `x` has been replaced by byte `b`.

```
unsigned replace_byte
  (unsigned x, int i, unsigned char b)
```

Here are some examples showing how the function should work

```
replace_byte(0x12345678, 2, 0xAB) --> 0x12AB5678
replace_byte(0x12345678, 0, 0xAB) --> 0x123456AB
```

Problem 2.23 [Category 3]:
Fill in code for the following C functions. Function `srl` performs a logical right shift using an arithmetic right shift (given by value `xsra`), followed by other operations not including right shifts or division. Function `sra` performs an arithmetic right shift using a logical right shift (given by value `xsrl`), followed by other operations not including right shifts or division. You may assume that `int`'s are 32-bits long.

```
unsigned srl(unsigned x, int k)
{
  /* Perform shift arithmetically */
  unsigned xsra = (int) x >> k;

}

int sra(int x, int k)
{
  /* Perform shift logically */
  int xsrl = (unsigned) x >> k;

}
```

Problem 2.24 [Category 2]:
Assume we are running code on a 32-bit machine using two's complement arithmetic for signed variables. The variables are declared and initialized as follows:

```
int x = foo();   /* Arbitrary value */
int y = bar();   /* Arbitrary value */
unsigned ux = x;
unsigned uy = y;
```

For each of the following C expressions, either 1) argue that it is true (i.e., evaluates to 1) for all values of `x` and `y`, or 2) give example values of `x` and `y` for which it is false (i.e., evaluates to 0.)

A. `(x >= 0) || ((2*x) < 0)`

B. `(x & 7) != 7 || (x<<30 < 0)`

C. `(x * x) >= 0`

D. `x < 0 || -x <= 0`

E. `x > 0 || -x >= 0`

F. `x*y == ux*uy`

G. `~x*y + uy*ux == -y`

2.4 Floating Point

Floating point representations provide a means of encoding rational numbers of the form $V = x \times 2^y$. It is useful for performing computations involving very large numbers ($|V| \gg 0$) or numbers very close to 0 ($|V| \ll 0$).

Up until the 1980's, every computer manufacturer devised its own conventions for how floating point numbers were represented and the details of the operations performed on them. In addition, they often did not worry too much about the accuracy of the operations, viewing speed and ease of implementation as being more critical than numerical precision.

All of this changed around 1985 with the advent of IEEE Standard 754, a carefully crafted standard for representing floating point numbers and the operations performed on them. This effort started in 1976 under Intel's sponsorship with the design of the 8087, a chip that provided floating point support for the 8086 processor. They hired Prof. William Kahan of University of California, Berkeley as a consultant to help design the standard and then pushed it through the standardization process with the Institute of Electrical and Electronics Engineers (IEEE). Nowadays virtually all computers support IEEE floating point. This has greatly improved the portability of scientific application programs across different machines.

2.4.1 Fractional Binary Numbers

A first step in understanding floating point numbers is to consider binary numbers having fractional values.

Let us first examine the more familiar decimal notation. Decimal notation uses a represention of the form: $d_m d_{m-1} \cdots d_1 d_0 . d_{-1} d_{-2} \cdots d_{-n}$, where each decimal digit d_i ranges between 0 and 9. This notation represents a number

$$d = \sum_{i=-n}^{m} 10^i \times d_i$$

The weighting of the digits is defined relative to the decimal point symbol '.': digits to the left are weighted by positive powers of ten, giving integral values, while digits to the right are weighted by negative powers of ten, giving fractional values. For example, 12.34_{10} represents the number $1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2} = 12\frac{34}{100}$.

By analogy, consider a notation of the form $b_m b_{m-1} \cdots b_1 b_0 . b_{-1} b_{-2} \cdots b_{-n}$, where each binary digit, or bit, b_i ranges between 0 and 1. This notation represents a number

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

The symbol '.' now becomes a *binary point*, with bits on the left being weighted by positive powers of two, and those on the right being weighed by negative powers of two. 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 two. For example, while 101.11_2 represents the number $5\frac{3}{4}$, 10.111_2 represents the number $2 + 0 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 2\frac{7}{8}$. Similarly, shifting the binary point one position to the right has the effect of multiplying the number by two, e.g., 1011.1_2 represents the number $8 + 0 + 2 + 1 + \frac{1}{2} = 11\frac{1}{2}$.

Note that numbers of the form $0.11 \cdots 1_2$ 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.

Assuming we consider only finite length encodings, decimal notation cannot represent numbers such as $\frac{1}{3}$ and $\frac{5}{7}$ exactly. Similarly, fractional binary notation can only represent numbers that can be written $x \times 2^y$. Other values can only be approximated. For example, although the number $\frac{1}{5}$ can be approximated with increasing accuracy by lengthening the binary representation, we cannot represent it exactly as a fractional binary number:

Representation	Value	Decimal
0.0_2	0	0.0_{10}
0.01_2	$\frac{1}{4}$	0.25_{10}
0.010_2	$\frac{2}{8}$	0.25_{10}
0.0011_2	$\frac{3}{16}$	0.1875_{10}
0.00110_2	$\frac{6}{32}$	0.1875_{10}
0.001101_2	$\frac{13}{64}$	0.203125_{10}
0.0011010_2	$\frac{26}{128}$	0.203125_{10}
0.00110011_2	$\frac{51}{256}$	0.19921875_{10}

Problem 2.25 [Category 1]:
Fill in the missing information in the table below

Fractional Value	Binary Rep.	Decimal Rep.
$\frac{1}{4}$	0.01	0.25
$\frac{3}{8}$		
$\frac{23}{16}$		
	10.1101	
	1.011	
		5.375
		3.0625

Problem 2.26 [Category 2]:
Consider numbers having a binary representation consisting of an infinite string of the form $0.y\,y\,y\,y\,y\,y\cdots$, where y is a k-bit sequence. For example, the binary representation of $\frac{1}{3}$ is $0.01010101\cdots$ ($y = 01$), while the representation of $\frac{1}{5}$ is $0.001100110011\cdots$ ($y = 0011$).

A. Let $Y = B2U_k(y)$, i.e., the number having binary representation y. Give a formula in terms of Y and k for the value represented by the infinite string. [Hint: Consider the effect of shifing the binary point k positions to the right.]

B. What is the numeric value of the string for the following values of y:

(a) 001
(b) 1001
(c) 000111

2.4.2 Floating Point Representation

Positional notation such as considered in the previous section would be very inefficient for representing very large numbers. For example, the represention of 5×2^{100} would consist of the bit pattern 101 followed by one hundred 0's. Instead, we would like to represent numbers in a form $x \times 2^y$ by giving the values of x and y.

The IEEE floating point 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 ($s = 0$.)
- 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 two.

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.

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.

The value encoded by a given bit representation can be divided into three different cases, depending on the value of `exp`.

Normalized Values

This is the most common case. They occur when the bit pattern of `exp` is neither all 0's (numeric value 0) or all 1's (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 $2^{k-1} - 1$ (i.e., 127 for single precision and 1023 for double.)

The fraction field `frac` is interpreted as representing the fractional value f, where $0 \leq f < 1$, having binary representation $0.f_{n-1} \cdots f_1 f_0$, i.e., 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 $1 \leq M < 2$. In that case, we do not need to explicitly represent the leading bit, since it always equals 1.

Denormalized Values

When the exponent field is all 0's, the represented number is in *denormalized* form. In this case, the exponent value is $E = 1 - Bias$ and the significand value is $M = f$, i.e., the value of the fraction field without an implied leading 1.

Denormalized numbers serve two purposes. First, they provide a way to represent numeric value 0, since with a normalized number we must always have $M \geq 1$, and hence we cannot represent 0. In fact the floating point representation of $+0.0$ has a bit pattern of all 0's: the sign bit is 0, the exponent field is all 0's (indicating a denormalized value), and the fraction field is all 0's, giving $M = f = 0$. Curiously, when the sign bit is 1, but the other fields are all 0's, we get the value -0.0. With IEEE floating point format, the values -0.0 and $+0.0$ are considered different in some ways and the same in others.

A second function of denormalized numbers is to represent numbers that are very close to 0.0. They provide a property known as *gradual underflow* in which possible numeric values are spaced evenly near 0.0.

Special Values

A final category of values occurs when the exponent field is all 1's. When the fraction field is all 0's, the resulting values represent infinity: either $+\infty$ when $s = 0$, or $-\infty$ when $s = 1$. Infinity can represent results that *overflow*, e.g., when we multiply two very large numbers, or when we divide by zero. When the fraction field is nonzero, the resulting value is called a "NaN", short for "Not a Number." Such values are returned as the result of an operation where the result cannot be given as a real number or as infinity, e.g., when computing $\sqrt{-1}$ or $\infty - \infty$.

2.4.3 Example Numbers

Table 2.12 shows some examples for a hypothetical 8-bit floating point format having $k = 4$ exponent bits and $n = 3$ fraction bits. The bias is $2^{4-1} - 1 = 7$. The table is divided into 3 regions representing the 3 classes of numbers. Closest to 0 are the denormalized numbers, starting with 0 itself. Denormalized numbers in this format have $E = 1 - 7 = -6$, giving a weight $2^E = \frac{1}{64}$. The fractions f range over the values $0, \frac{1}{8}, \ldots, \frac{7}{8}$, giving numbers V in the range 0 to $\frac{7}{8 \times 64} = \frac{7}{512}$.

The smallest normalized numbers in this format also have $E = 1 - 7 = -6$, and the fractions also range over the values $0, \frac{1}{8}, \ldots \frac{7}{8}$. However, the significands then range from $1 + 0 = 1$ to $1 + \frac{7}{8} = \frac{15}{8}$, giving numbers V in the range $\frac{8}{512}$ to $\frac{15}{512}$.

Observe the smooth transition between the largest denormalized number $\frac{7}{512}$ and the smallest normalized number $\frac{8}{512}$. This smoothness is due to our definition of E for denormalized values. By making it $1 - Bias$ rather than $-Bias$, we compensate for the fact that the significand of a denormalized number does not have an implied leading 1.

As we increase the exponent, we get successively larger normalized values, passing through 1.0 and then to the largest normalized number. This number has exponent $E = 7$, giving a weight $2^E = 128$. The fraction equals $\frac{7}{8}$ giving a signficand $M = \frac{15}{8}$. Thus the numeric value is $V = 240$. Going beyond this overflows to $+\infty$.

Description	Bit Rep.	e	E	f	M	V
Zero	`0 0000 000`	0	-6	0	0	0
Smallest Pos.	`0 0000 001`	0	-6	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{512}$
	`0 0000 010`	0	-6	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{2}{512}$
	`0 0000 011`	0	-6	$\frac{3}{8}$	$\frac{3}{8}$	$\frac{3}{512}$
	…					
	`0 0000 110`	0	-6	$\frac{6}{8}$	$\frac{6}{8}$	$\frac{6}{512}$
Largest Denorm.	`0 0000 111`	0	-6	$\frac{7}{8}$	$\frac{7}{8}$	$\frac{7}{512}$
Smallest Norm.	`0 0001 000`	1	-6	0	$\frac{8}{8}$	$\frac{8}{512}$
	`0 0001 001`	1	-6	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{512}$
	…					
	`0 0110 110`	6	-1	$\frac{6}{8}$	$\frac{14}{8}$	$\frac{14}{16}$
	`0 0110 111`	6	-1	$\frac{7}{8}$	$\frac{15}{8}$	$\frac{15}{16}$
One	`0 0111 000`	7	0	0	$\frac{8}{8}$	1
	`0 0111 001`	7	0	$\frac{1}{8}$	$\frac{9}{8}$	$\frac{9}{8}$
	`0 0111 010`	7	0	$\frac{2}{8}$	$\frac{10}{8}$	$\frac{10}{8}$
	…					
	`0 1110 110`	14	7	$\frac{6}{8}$	$\frac{14}{8}$	224
Largest Norm.	`0 1110 111`	14	7	$\frac{7}{8}$	$\frac{15}{8}$	240
Infinity	`0 1111 000`	–	–	–	–	$+\infty$

Table 2.12: **Example Nonnegative Values for 8-bit Floating Point Format.** There are $k = 4$ exponent bits and $n = 3$ significand bits. The bias is 7.

One interesting property of this representation is that if we interpret the bit representations of the values in Table 2.12 as unsigned integers, they occur in ascending order, as do the values they represent as floating point numbers. This is no accident—the IEEE format was designed so that floating point numbers could be sorted using an integer sorting routine. A minor difficulty is in dealing with negative numbers, since they have a leading one, and they occur in descending order, but this can be overcome without requiring floating point operations to perform comparisons.

Problem 2.27 [Category 1]:
Fill in return value for the following procedure that tests whether its first argument is greater than or equal to its second. Assume the function `f2u` returns an unsigned 32-bit number having the same bit representation as its floating point argument.

```
int float_ge(float x, float y)
{
  unsigned ux = f2u(x);
  unsigned uy = f2u(y);
  /* Get the sign bits */
  unsigned sx = ux >> 31;
  unsigned sy = uy >> 31;
  /* Give an expression using only values
     ux, uy, sx, and sy
  */

  return

}
```

Problem 2.28 [Category 1]:
Consider a 5-bit floating point representation based on the IEEE floating point format, with one sign bit, two exponent bits ($k = 2$), and two fraction bits ($n = 2$). The exponent bias is $2^{2-1} - 1 = 1$.
The table below enumerates the entire non-negative range for this 5-bit floating point representation. Fill in the blank table entries using the following directions:

e**:** The value represented by considering the exponent field to be an unsigned integer.

E**:** The value of the exponent after biasing.

f**:** The value of the fraction.

M**:** The value of the significand.

V**:** The numeric value represented.

Express the values of f, M and V as fractions of the form $\frac{x}{4}$. You need not fill in entries marked "—".

Bits	e	E	f	M	V
`0 00 00`					
`0 00 01`					
`0 00 10`					
`0 00 11`					
`0 01 00`					
`0 01 01`					
`0 01 10`					
`0 01 11`					
`0 10 00`	2	1	$\frac{0}{4}$	$\frac{4}{4}$	$\frac{8}{4}$
`0 10 01`					
`0 10 10`					
`0 10 11`					
`0 11 00`	—	—	—	—	$+\infty$
`0 11 01`	—	—	—	—	NaN
`0 11 10`	—	—	—	—	NaN
`0 11 11`	—	—	—	—	NaN

Table 2.13 shows the representations and numeric values of some important single and double precision floating point numbers. As with the 8-bit format shown in Table 2.12 we can see some general properties for a floating point representation with a k-bit exponent and an n-bit fraction:

- The value $+0.0$ always has a bit representation of all 0's.

Description	exp	frac	Single Precision		Double Precision	
			Value	Decimal	Value	Decimal
Zero	$00\cdots00$	$0\cdots00$	0	0.0	0	0.0
Smallest denorm.	$00\cdots00$	$0\cdots01$	$2^{-23}\times2^{-126}$	1.4×10^{-45}	$2^{-52}\times2^{-1022}$	4.9×10^{-324}
Largest denorm.	$00\cdots00$	$1\cdots11$	$(1-\epsilon)\times2^{-126}$	1.2×10^{-38}	$(1-\epsilon)\times2^{-1022}$	2.2×10^{-308}
Smallest norm.	$00\cdots01$	$0\cdots00$	1×2^{-126}	1.2×10^{-38}	1×2^{-1022}	2.2×10^{-308}
One	$01\cdots11$	$0\cdots00$	1×2^{0}	1.0	1×2^{0}	1.0
Largest norm.	$11\cdots10$	$1\cdots11$	$(2-\epsilon)\times2^{127}$	3.4×10^{38}	$(2-\epsilon)\times2^{1023}$	1.8×10^{308}

Table 2.13: **Examples of Nonnegative Floating Point Numbers.**

- The smallest positive denormalized value has a bit representation consisting of a 1 in the least significant bit position and otherwise all 0's. It has a fraction (and significand) value $M = f = 2^{-n}$ and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = 2^{-n-2^{k-1}+2}$.

- The largest denormalized value has a bit representation consisting of an exponent field of all 0's and a fraction field of all 1's. It has a fraction (and significand) value $M = f = 1 - 2^{-n}$ (which we have written $1 - \epsilon$) and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = (1 - 2^{-n}) \times 2^{-2^{k-1}+2}$, which is just slightly smaller than the smallest normalized value.

- The smallest positive normalized value has a bit representation with a 1 in the least significant bit of the exponent field and otherwise all 0's. It has a significand value $M = 1$ and an exponent value $E = -2^{k-1} + 2$. The numeric value is therefore $V = 2^{-2^{k-1}+2}$.

- The value 1.0 has a bit representation with all but the most significant bit of the exponent field equal to 1 and all other bits equal to 0. Its significand value is $M = 1$ and its exponent value is $E = 0$.

- The largest normalized value has a bit representation with a sign bit of 0, the least significant bit of the exponent equal to 0, and all other bits equal to 1. It has a fraction value of $f = 1 - 2^{-n}$, giving a significand $M = 2 - 2^{-n}$ (which we have written $2 - \epsilon$.) It has an exponent value $E = 2^{k-1} - 1$, giving a numeric value $V = (2 - 2^{-n}) \times 2^{2^{k-1}-1} = (1 - 2^{-n-1}) \times 2^{2^{k-1}}$.

Problem 2.29 [Category 1]:
Given a floating point format with a k-bit exponent and an n-bit fraction, give formulas for the exponent E, significand M, the fraction f, and the value V for the following quantities. In addition, describe the bit representation.

A. The number 5.0.

B. The largest odd integer that can be represented exactly.

C. The reciprocal of the smallest positive normalized value.

Problem 2.30 [Category 1]:

A. For a floating point format with a k-bit exponent and an n-bit fraction, give a formula for the smallest positive integer that cannot be represented exactly.

B. What is the numeric value of this integer for single precision format ($k = 8, n = 23$)?

Problem 2.31 [Category 1]:
Intel architecture's also support an "extended precision" floating point format with an 80-bit word divided into a sign bit, $k = 15$ exponent bits, a single *integer* bit, and $n = 63$ fraction bits. The integer bit corresponds to the implied bit in the IEEE floating point representation, i.e., it equals 1 for normalized values and equals 0 for denormalized values. Fill in the following table giving the approximate values of some "interesting" numbers in this format:

Description	Extended Precision	
	Value	Decimal
Smallest denormalized		
Smallest normalized		
Largest normalized		

Problem 2.32 [Category 1]:
Consider a 16-bit floating point representation based on the IEEE floating point format, with one sign bit, seven exponent bits ($k = 7$), and eight fraction bits ($n = 8$). The exponent bias is $2^{7-1} - 1 = 63$.
Fill in the table below for the following numbers, with the following instructions for each column:

Hex: The 4 hexadecimal digits describing the encoded form.

M**:** The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer, and y is an integral power of 2. Examples include: 0, $\frac{67}{64}$, and $\frac{1}{256}$.

E**:** The integer value of the exponent.

V**:** The numeric value represented. Use the notation x or $x \times 2^z$, where x and z are integers.

As an example, to represent the number $\frac{7}{2}$, we would have $s = 0$, $M = \frac{7}{4}$, and $E = 1$. Our number would therefore have an exponent field of `0x40` (decimal value $63 + 1 = 64$) and a significand field `0xC0` (binary 11000000_2), giving a hex representation `40C0`.
You need not fill in entries marked "—".

Description	Hex	M	E	V
-0				—
Smallest value > 1				
256				—-
Largest Denormalized				
$-\infty$		—	—	—
Number with hex representation `3AA0`	—			

Problem 2.33 [Category 1]:
You have been assigned the task of writing a C function to compute a floating point representation of 2^x. You realize that the best way to do this is to directly construct the IEEE single precision representation of the result. When x is too small, your routine will return 0.0. When x is too large, it will return $+\infty$. Fill in the blank portions of the following code to compute the correct result. Assume the function `u2f` returns a floating point value having an identical bit representation as its unsigned argument.

```
/* Compute 2**x */
float fpwr2(int x) {

  /* Result exponent and significand */
  unsigned exp, sig;
  unsigned u;

  if (x < ______) {
    /* Too small.  Return 0.0 */
    exp = ____________;
    sig = ____________;
  } else if (x < ______) {
    /* Denormalized result */
    exp = ____________;
    sig = ____________;
  } else if (x < ______) {
    /* Normalized result. */
    exp = ____________;
    sig = ____________;
  } else {
    /* Too big.  Return +oo */
    exp = ____________;
    sig = ____________;
  }

  /* Pack exp and sig into 32 bits */
  u = exp << 23 | sig;
  /* Return as float */
  return u2f(u);
}
```

Mode	\$1.40	\$1.60	\$1.50	\$2.50	\$–1.50
Round-To-Even	\$1	\$2	\$2	\$2	\$–2
Round-Toward-Zero	\$1	\$1	\$1	\$2	\$–1
Round-Down	\$1	\$1	\$1	\$2	\$–2
Round-Up	\$2	\$2	\$2	\$3	\$–1

Table 2.14: **Illustration of Rounding Modes for Dollar Rounding.** The first rounds to a nearest value, while the other three bound the result above or below.

Problem 2.34 [Category 1]:
Around 250 B.C., the Greek mathematician Archimedes proved that $\frac{223}{7} < \pi < \frac{22}{7}$. Had he had access to a computer and the standard library `<math.h>` he would have been able to determine that the single precision approximation of π has hexadecimal representation `0x40490FDB`. Of course, all of these are just approximations, since π is not rational.

A. What is the fractional binary number denoted by this floating point value?

B. What is the fractional binary representation of $\frac{22}{7}$? [Hint: See Problem 2.26].

C. At what bit position (relative to the binary point) do these two approximations to π diverge?

2.4.4 Rounding

Floating point arithmetic can only approximate real arithmetic, since the representation has limited range and precision. Thus, for a value x, we generally want a systematic method of finding the "closest" matching value x' that can be represented in the desired floating point format. This is the task of the *rounding* operation. The key problem is to define the direction to round a value that is halfway between two possibilities. For example, if I have \$1.50 and want to round it to the nearest dollar, should the result be \$1 or \$2? An alternative approach is to maintain a lower and an upper bound on the actual number. For example, we could determine representable values x^- and x^+ such that the value x is guaranteed to lie between them: $x^- \leq x \leq x^+$. The IEEE floating point format defines four different *rounding modes*. The default method finds a closest match, while the other three can be used for computing upper and lower bounds.

Table 2.14 illustrates the four rounding modes applied to the problem of rounding a monetary amount to the nearest whole dollar. Round-to-even is the default mode. It attempts to find a closest match. Thus, it rounds \$1.40 to \$1 and \$1.60 to \$2, since these are the closest whole dollar values. The only design decision is to determine the effect of rounding values that are halfway between two possible results. Round-to-Even mode adopts the convention that it rounds the number either upward or downward such that the least significant digit of the result is even. Thus, it rounds both \$1.50 and \$2.50 to \$2.

The other three modes produce guaranteed bounds on the actual value. These can be useful in some numerical applications. Round-Toward-Zero mode rounds positive numbers downward and negative numbers upward, giving a value $\hat{x}$ such that $|\hat{x}| \leq |x|$. Round-Down mode rounds both positive and negative numbers downward, giving a value x^- such that $x^- \leq x$. Round-Up mode rounds both positive and negative numbers upward, giving a value x^+ such that $x \leq x^+$.

Round-to-even at first seems like it has a rather arbitrary goal—why is there any reason to prefer even numbers? Why not consistently round values halfway between two representable values upward? The problem with such a convention is that one can easily imagine scenarios in which rounding a set of data values would then introduce a statistical bias into the computation of an average of the values. The average of a set of numbers that we rounded by this means would be slightly higher than the average of the numbers themselves. Conversely, if we always rounded numbers halfway between downward, the average of a set of rounded numbers would be slightly lower than the average of the numbers themselves. Rounding toward even numbers avoids this statistical bias in most real-life situations. It will round upward about 50% of the time and round downward about 50% of the time.

Round-to-even rounding can be applied even when we are not rounding to a whole number. We simply consider whether the least significant digit is even or odd. For example, suppose we want to round decimal numbers to the nearest hundredth. We would round 1.2349999 to 1.23 and 1.2350001 to 1.24, regardless of rounding mode, since they are not halfway between 1.23 and 1.24. On the other hand, we would round both 1.2350000 and 1.2450000 to 1.24, since 4 is even.

Similarly, round-to-even rounding can be applied to binary fractional numbers. We consider least significant bit value 0 to be even and 1 to be odd. In general, the rounding mode is only significant when we have a bit pattern of the form $XX \cdots X.YY \cdots Y100 \cdots$, where X and Y denote arbitary bit values with the rightmost Y being the position to which we wish to round. Only bit patterns of this form denote values that are halfway between two possible results. As examples, consider the problem of rounding values to the nearest quarter (i.e., 2 bits to the right of the binary point). We would round 10.00011_2 ($2\frac{3}{32}$) down to 10.00_2 (2), and 10.00110_2 ($2\frac{3}{16}$) up to 10.01_2 ($2\frac{1}{4}$), because these values are not halfway between two possible values. We would round 10.11100_2 ($2\frac{7}{8}$) up to 11.00_2 (3) and 10.10100_2 down to 10.10_2 ($2\frac{1}{2}$) since these values are halfway between two possible results, and we prefer to have the least significant bit equal to zero.

2.4.5 Floating Point Operations

The IEEE standard specifies a simple rule for determining the result of an arithmetic operation such as addition or multiplication. Viewing floating point values x and y as real numbers, and some operation $\odot$ defined over real numbers, the computation should yield $Round(x \odot y)$, i.e., the result of applying rounding to the exact result of the real operation. In practice, there are clever tricks floating point unit designers use to avoid performing this exact computation, since the computation need only be sufficiently precise to guarantee a correctly rounded result. When one of the arguments is a special value such as -0, ∞ or NaN, the standard specifies conventions that attempt to be reasonable. For example $1/-0$ is defined to yield $-\infty$, while $1/+0$ is defined to yield $+\infty$.

One strength of the IEEE standard's method of specifying the behavior of floating point operations is that it is independent of any particular hardware or software realization. Thus, we can examine its abstract mathematical properties without considering how it is actually implemented.

We saw earlier that integer addition, both unsigned and two's complement, forms an Abelian group. Addition over real numbers also forms an Abelian group, but we must consider what effect rounding has on these properties. Let us define $x +_f y$ to be $Round(x + y)$. This operation is defined for all values of x and y, although it may yield infinity even when both x and y are real numbers due to overflow. The operation is commutative, with $x +_f y = y +_f x$ for all values of x and y. On the other hand, the operation is not associative. For example, with single precision floating point the expression `(3.14+1e10)-1e10` would evaluate to `0.0`—the value 3.14 would be lost due to rounding. On the other hand, the expression `3.14+(1e10-1e10)` would evaluate to `3.14`. As with an Abelian group, most values have inverses under floating point addition, i.e., $x +_f -x = 0$. The exceptions are infinities (since $+\infty - \infty = NaN$), and NaN's, since $NaN +_f x = NaN$ for any x.

The lack of associativity in floating point addition is the most important group property that is lacking. It has important implications for scientific programmers and compiler writers. For example, suppose a compiler is given the following code fragment:

```
x = a + b + c;
y = b + c + d;
```

The compiler might be tempted to save one floating point addition by generating the code:

```
t = b + c;
x = a + t;
y = t + d;
```

However, this computation might yield a different value for `x` than would the original, since it uses a different association of the addition operations. In most applications, the difference would be so small as to be inconsequential. Unfortunately, compilers have no way of knowing what trade-offs the user is willing to make between efficiency and faithfulness to the exact behavior of the original program. As a result, they tend to be very conservative, avoiding any optimizations that could have even the slightest effect on functionality.

On the other hand, floating point addition satisfies the following monotonicity property: if $a \geq b$ then $x + a \geq x + b$ for any values of a, b, and x other than NaN. This property of real (and integer) addition is not obeyed by unsigned or two's complement addition.

Floating point multiplication also obeys many of the properties one normally associates with multiplication, namely those of a ring. Let us define $x *_f y$ to be $Round(x \times y)$. This operation is closed under multiplication (although possibly yielding infinity or NaN), it is commutative, and it has 1.0 as a multiplicative identity. On the other hand, it is not associative due to the possibility of overflow or the loss of precision due to rounding. For example, with single precision floating point, the expression `(1e20*1e20)*1e-20` will evaluate to $+\infty$, while `1e20*(1e20*1e-20)` will evaluate to `1e20`. In addition, floating point multiplication does not distribute over addition. For example, with single precision floating point, the expression `1e20*(1e20-1e20)` will evaluate to `0.0`, while `1e20*1e20-1e20*1e20` will evaluate to `NaN`.

On the other hand, floating point multiplication satisfies the following monotonicity properties for any values of a, b, and c other than NaN:

$$a \geq b \text{ and } c \geq 0 \Rightarrow a *_f c \geq a *_f c$$
$$a \geq b \text{ and } c \leq 0 \Rightarrow a *_f c \leq a *_f c$$

In addition, we are also guaranteed that $a *_f a \geq 0$, as long as $a \neq NaN$. As we saw earlier, none of these monotonicity properties hold for unsigned or two's complement multiplication.

This lack of associativity and distributivity is of serious concern to scientific programmers and to compiler writers. Even such a seemingly simple task as writing code to determine whether two lines intersect in three-dimensional space can be a major challenge.

2.4.6 Floating Point in C

C provides two different floating point data types: `float` and `double`. On machines that support IEEE floating point, these data types correspond to single and double precision floating point. In addition, the machines use the Round-To-Even rounding mode. Unfortunately, since the C standard does require the machine use IEEE floating point, there are no standard methods to change the rounding mode or to get special values such as -0, $+\infty$, $-\infty$, or NaN.

> **Problem 2.35** [Category 1]:
> Write macro definitions for a machine with IEEE floating point that will generate the following double precision values:
>
> A. $+\infty$: `#define POS_INFINITY` __________
>
> B. $-\infty$: `#define NEG_INFINITY` __________
>
> C. -0: `#define NEG_ZERO` __________
>
> Recall that the largest finite number that can be represented with double precision is around 1.8×10^{308}.

When casting values between `int`, `float`, and `double` formats, the program changes the numeric values and the bit representations as follows:

- From `int` to `float`, the number cannot overflow, but it may be rounded.
- From `int` or `float` to `double`, the exact numeric value can be preserved because `double` has both greater range (i.e., the range of representable values), as well as greater precision (i.e., the number of significant bits.)
- From `double` to `float`, the value can overflow to $+\infty$ or $-\infty$, since the range is smaller. Otherwise it may be rounded since the precision is smaller.
- From `float` or `double` to `int` the value will be truncated toward zero. For example 1.999 will be converted to 1, while -1.999 will be converted to -1. Note that this behavior is very different from rounding. In addition, the value may overflow. The C standard does not specify a fixed result for this case, but on most machines the result will either be $TMax_w$ or $TMin_w$, where w is the number of bits in an `int`.

Problem 2.36 [Category 1]:
Assume variables `x`, `f`, and `d` are of type `int`, `float`, and `double`, respectively. Their values are arbitrary, except that neither `f` nor `d` equals $+\infty$, $-\infty$, or NaN. For each of the following C expressions, either argue that it will always be true (i.e., evaluate to 1) or give a value for the variables such that it is not true (i.e., evaluates to 0).

A. `x == (int)(float) x`

B. `x == (int)(double) x`

C. `f == (float)(double) f`

D. `d == (float) d`

E. `f == -(-f)`

F. `2/3 == 2/3.0`

G. `(d >= 0.0) || ((d*2) < 0.0)`

H. `(d+f)-d == f`